VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Aproximace statistických kompresních metod pro zlepšení Lazy Evaluation

Approximation of Statistical Compression for Improvement of Lazy Evaluation

2015 Bc. Eva Čecháková

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Eva Čecháková**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Aproximace statistických kompresních metod pro zlepšení Lazy Evaluation
Approximation of Statistical Compression for Improvement of Lazy Evaluation

Zásady pro vypracování:

Cílem práce je vylepšit metodu vylepšující Lazy Evaluation, navrženou v rámci bakalářské práce, o možnost zapojení statistické komprese. Pozornost bude zejména věnována efektivnímu zapojení statistické komprese v celkovém běhu algoritmu tak, aby nebylo nutné neustálé přepočítávání statistických modelů.

Práce bude obsahovat:
1. Popis metody Lazy Evaluation v LZ77 kompresi a místa pro zapojení statistických kompresních algoritmů.
2. Popis existujících metod pro aproximaci statistických kompresních algoritmů.
3. Návrh řešení definovaného problému.
4. Testování implementace a provedení experimentů.

Seznam doporučené odborné literatury:

[1] Data Compression: The Complete Reference, David Salomon, 4. ed., Springer, 2007

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2015

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA 1

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Opavě 30. července 2015 …….

Poděkování

Ráda bych tímto poděkovala panu doc. Ing. Janu Platošovi, Ph.D., za odborné vedení při vytváření mé diplomové práce a mé rodině za podporu a trpělivost.

Abstrakt

Předmětem diplomové práce je vylepšení metody lazy evaluation o možnosti zapojení statistické komprese ve smyslu zlepšení kompresního poměru. Po teoretické stránce se práce zaměřuje na princip a popis algoritmů, které jsou využívány při implementaci a testování, důraz je kladen na metodu LZSS s využitím lazy evaluation a Huffmanova kódování a její optimalizovanou variantu. Praktická část práce aplikuje návrh řešení pomocí jazyka C# v prostředí Visual Studio. Testování probíhá nad testovacími soubory z kolekcí The Canterbury Corpus a The Calgary Corpus a ověřuje efektivitu navrženého algoritmu LZSS Huffman optimal. Získané poznatky jsou pak interpretovány pomocí grafů výsledků komprese a tabulek porovnání jednotlivých metod nad různými soubory.

Klíčová slova

komprese dat, bezztrátová komprese, komprese textu, aproximace, LZ77, LZSS, LZH, deflate opožděné vyhodnocování, Huffmanovo kódování, optimální parsování

Abstract

This thesis deals with the improvement of the Lazy Evaluation method with possibilities of integrating static compression in the sense of bettering the compression ratio. Theoretical part of the work focuses on the principle and description of algorithms, which are used in implementation and testing. The stress is laid on the LZSS method with the use of lazy evaluation and Huffman coding and its optimized version. Practical part of the work applies solution proposal with the help of the C# language in the Visual Studio environment. The testing takes place over the testing files from The Canterbury Corpus and The Calgary Corpus collections and verifies the efficiency of the projected LZSS Huffman optimal algorithm. The gained knowledge is then interpreted by compression results charts and tables of comparison of the individual methods over various files.

Key words

data compression, lossless compression, text compression, approximation, LZ77, LZSS, LZH, deflate, lazy evaluation, Huffman coding, optimal parsing

Seznam použitých zkratek a symbolů

EOF	Konec souboru (End Of File)

Obsah

Seznam tabulek

Seznam obrázků

1 Úvod

V dnešní době dochází k velkému shromažďování a uchovávání dat, ať už se jedná o data textová, zvuková, obrazová či video soubory. Přes velmi rychle rozvíjející se odvětví komunikačních technologií je nutností i jakýmsi trendem používat a neustále vyvíjet kompresní programy, které zmenšují velikost těchto dat. Tyto programy slouží v praxi ke sloučení více souborů do jednoho – archivaci, ale i rozdělení souboru na více souborů o dané velikosti. Komprese se používá pro přenos a ukládaní dat.

Principem komprese je snížení počtu bitů, které reprezentují datový objekt. Algoritmy komprese spočívají v nalezení a odstranění redundantních dat, aniž by došlo ke ztrátě důležitých informací. U textových dat je důležitá každá informace, proto se pro takováto data používá komprese bezztrátová. Naopak u dat zvukových či obrazových je ztrátovost často principem komprese, odstraňuje především informace, které jsou vzhledem k nedokonalosti lidských smyslů nepotřebné. Vždy je nutné definovat přesný postup dané komprese. Součástí tohoto postupu musí být i opačný postup, nezbytný k rekonstrukci původních dat.

1.1 Cíle práce

Tato diplomová práce se zabývá vylepšením optimalizovaných kompresních metod LZ77 a LZSS s využitím lazy evaluation, které byly navrženy v rámci bakalářské práce, o možnost zapojení statistické komprese v celkovém běhu algoritmu tak, aby nebylo nutné neustále přepočítávání statistických modelů.

Přes teoretický popis dané problematiky se dostaneme k návrhu řešení, jeho implementaci a následném testování nad vybranou kolekcí datový souborů. Cílem práce je také zhodnocení výsledků optimalizovaného zapojení statistické komprese s využitím aproximace do metody navržené v rámci bakalářské práce a srovnání s existujícími kompresními algoritmy.

1.2 Obsah práce

Práce se dělí na dvě základní části teoretickou a praktickou, praktická část je detailněji rozdělena na:

- návrh řešení zapojení statistické komprese a její aproximace
- implementace návrhu
- testování
  - analýza vstupních dat
  - testování a provádění experimentů
  - zhodnocení výsledků

Po úvodní kapitole následuje vymezení základních pojmů spojených s obecnou kompresí dat. Velká část pozornosti (kapitoly 3 až 7) je věnována teoretickému rozboru pro práci stěžejních kompresních metod LZ77, LZSS, Huffmanova kódování, LZH, Deflate a Fibonacciho kódování. Kapitoly Lazy evaluation a Optimální parsování se pak zaměřují na

využití opožděného vyhodnocování a jeho optimalizaci pro některé z výše uvedených kompresí. Jádrem práce je kapitola 10, kde je popsána problematika zapojení Huffmanova kódování do LZSS využívajícího lazy evaluation a návrh jejího řešení. Dále si v této kapitole představíme existující metody pro možnou aproximaci statistických kompresních algoritmů. Následuje kapitola s krátkým popisem implementace navrženého algoritmu. Důležitá je bezesporu kapitola Testování, která se věnuje analýze vstupních dat z kolekcí The Canterbury Corpus a The Calgary Corpus. Na základě výsledků získaných prováděním experimentů zhodnotíme přínos navrženého algoritmu. Nedílnou součástí práce jsou grafy a tabulky s výsledky testů, které podrobněji popisují efektivitu a srovnání jednotlivých algoritmů z hlediska kompresního poměru. Poslední kapitolou je závěr, kde jsou zhodnoceny všechny poznatky a přínosy této práce. Zmíněna je i možnost budoucího rozšíření a vylepšení navrženého algoritmu.

2 Komprese dat

Komprese (neboli komprimace či zhušťování dat) je speciální druh kódování dat za účelem zmenšení jejich objemu odstraněním nadbytečné informace (redundance) [6]. Toto kódování je určeno daným kompresním algoritmem, pomocí kterého jsme schopni jasně definovat postup komprese i zpětný postup dekomprese pro rekonstrukci původních dat. Kompresní algoritmus převádí *zdrojové jednotky* (symboly a posloupnosti symbolů – takzvaná slova a posloupnosti slov) na *posloupnosti bitů* [6]. Zdrojové jednotky se mohou označovat jako *vzory* (originály) a výsledné posloupnosti jako *obrazy* [6].

Důvodů, proč komprimovat data a tím zmenšovat jejich velikost, je mnoho, například pro archivaci souborů, přenos těchto souborů přes síť s omezenou rychlostí nebo kvůli omezené datové propustnosti.

Na vyjádření efektivnosti metod komprimace můžeme použít *komprimační (kompresní) poměr*, který je definován následovně:

$$komprimační\ poměr = \frac{\text{velikost obrazu}}{\text{velikost vzoru}},$$

kde například hodnota 0,8 znamená, že údaje pro komprimaci zabírají 80 % původního paměťového prostoru. Hodnoty větší než jedna znamenají negativní, obvykle nežádoucí expanzi zpracovaných dat [5].

2.1 Historie komprese

Významným mezníkem v historii komprese dat je rozvoj telekomunikací. Komprese dat byla nezbytná pro využití digitálního (nespojitého) signálu.

Proto nás nepřekvapí, že prvním použitím kódování je Morseova abeceda využívaná v telegrafii. Když se podrobněji podíváme na tabulku znaků této abecedy, vidíme, že písmena, která se nejčastěji využívají, jsou reprezentována menším počtem bitů (v angličtině jde o písmena „e“ a „t)

Statickou pravděpodobnost výskytu znaků ve zprávě využili později ve 40. letech 20. století Claude Shannon a Robert Fano. Velkým mezníkem je ovšem v roce 1951 Huffmanovo kódování, které je založeno na vytváření minimálního prefixového kódu. V 70. letech se používá i přes svou výpočetní náročnost adaptivní Huffmanovo kódování. Rok 1977 a Abraham Lempel nám přinesli kompresi pomocí ukazatelů, o rok později tuto metodu vylepšil Jacob Ziv a vznikly tak známé algoritmy jako LZ77 a LZ78, základ pro AR, ARJ, GZIP či ACE. 80. léta přinesla metodu LZW (Lempel-Zip-Welsch) pro formát GIF. Masivní rozšíření multimédií, digitalizace obrazu, zvuku i videa v 90. letech mělo za následek rozvoj kompresních ztrátových metod, jako je JPEG. V dnešní době stále zlepšující se výpočetní schopnosti počítačů umožňují použít složitější matematické metody a dosáhnout ještě zajímavějších výsledků. [8]

2.2 Druhy komprese

Základní vlastností rozdělující kompresní metody na dvě velké skupiny je ztrátovost. Ta určuje, zda při procesu komprese došlo k menší ztrátě dat či nikoliv.

2.2.1 Ztrátová

Ztrátová komprese je založena na odstraňování nepotřebných informací (detailů). Používá se hlavně pro kompresi zvukových, obrazových a video souborů, kde využívá nedokonalosti lidského zraku a sluchu. V praxi to znamená, že například u zvukových souborů se odstraňují signály mimo slyšitelný frekvenční rozsah nebo slabší signály, které jsou překrývány silnějšími. U obrazových souborů je možné kompresí snížit barevnou hloubku nebo rozlišení podle potřeby. Pro video soubory se často používají kombinace předešlých kompresních metod pro obraz a zvuk. Patří zde například: MPEG, JPEG, JPEG 2000 MP3, WMA, AAC, Vorbis.

2.2.2 Bezztrátová

Základní vlastností bezztrátové komprese je totožnost původních dat s daty dekomprimovanými – vždy zachovává kompletní informaci. Využívá se především u textových a binárních souborů nebo u vektorové grafiky.

Dle přístupu ke kompresi můžeme tuto skupinu dále dělit na:

- **Statistické algoritmy** – stanovuje se pravděpodobnost pro každý symbol ze vstupu a podle toho se vytváří co nejoptimálnější kód [2].
- **Slovníkové algoritmy** – nahrazuje často se vyskytující posloupnosti znaků, které ukládá do slovníku, za odpovídající kódy (kódová slova).
    - **Statické** – během procesu komprese se slovník nemění.
    - **Semiadaptivní** – slovník se během komprese mění (vytváří) podle aktuálně komprimovaných dat a je jejich součástí pro pozdější dekompresi.
    - **Adaptivní** – stejný postup jako u semiadaptivních algoritmů, pouze slovník není součástí komprimovaných dat, protože při dekomprimaci lze tento slovník opětovně vytvořit. Zde řadíme algoritmus LZ77 a jeho varianty.

2.2.3 Další rozdělení

Dle výpočetní náročnosti:

- Symetrické
- Asymetrické

Dle zpracování vstupních dat:

- Proudové
- Blokové

Dle počtu průchodů:

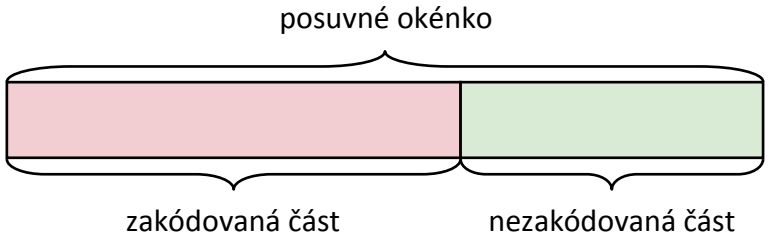
- Jedno-průchodové
- Více-průchodové

3 LZ77

Bezztrátový kompresní algoritmus LZ77, někdy také označován jako LZ1 ([1]), byl pojmenován podle Abrahama Lempela a Jacoba Ziva, kteří ho publikovali v roce 1977 ([9]). Řadí se mezi slovníkové algoritmy a v současnosti je jedním z nejpoužívanějších kompresních algoritmů.

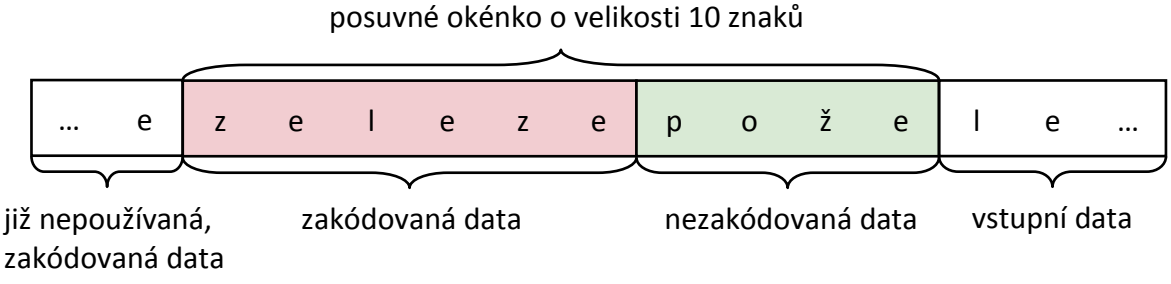
3.1 Princip

Algoritmus LZ77 využívá takzvaného posuvného okénka neboli anglicky sliding window. Při komprimaci se nezakódovanými daty (textem) posouvá pomyslné okénko. Toto okénko se skládá ze dvou částí, zakódovaná (anglicky search buffer) a nezakódovaná část (anglicky look-ahead buffer), znázorněno na obrázku 1.



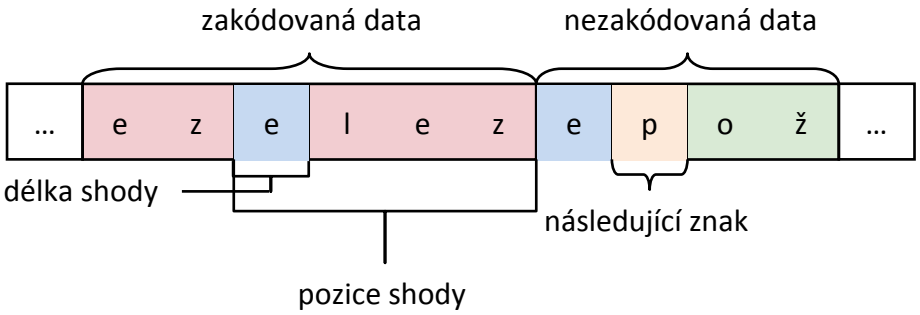
Obrázek 1 – Posuvné okénko

Z obrázku je patrné, že pro popis okénka jsou důležité dva parametry, velikost okénka a velikost nezakódované části. Velikost okénka je různá, většinou se setkáme s velikostí od 2048 bytů až po několik MB. U nezakódované části se velikost pohybuje v několikanásobně menším rozmezí, mezi 8 až 256 znaky [2]. Algoritmus pracuje pouze s daty v okénku, a tak paměťová náročnost algoritmu u různé délky vstupního řetězce zůstává konstantní, jak lze vidět na obrázku 2.



Obrázek 2 – Zaplněné posuvné okénko

Samotný princip algoritmu spočívá v prohledávání zakódované části směrem zprava doleva a nalezení co nejdelší shody pro sekvenci znaků z nezakódované části. Výsledkem je *trojice* parametrů, která je odesílána na výstup – *(pozice, délka, následující znak)*, kde *pozice* je vzdálenost nalezené shody (shodných znaků) od konce zakódované části, *délka* vyjadřuje počet znaků nalezené shody a *následující znak* je prvním znakem v nezakódované části, který již není obsažen ve znacích, pro které byla nalezena shoda, viz obrázek 3, kde výsledná trojice bude vypadat takto – (4, 1, "p").

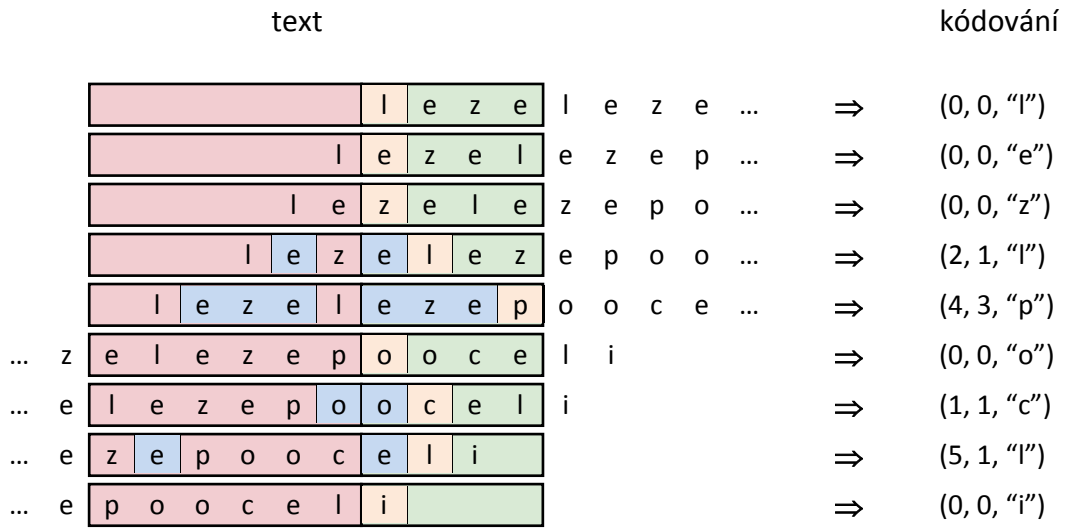


Obrázek 3 – Kódová trojice

Důvodem pro uvádění třetího parametru (následující znak) v trojici je situace, kdy pro daný znak z nezakódované části nebyl nalezen žádný shodný znak v zakódované části. V tomto případě se pozice i délka nastaví na hodnotu nula a třetí parametr – následující znak je nastaven na tento znak – (0, 0, znak) [4].

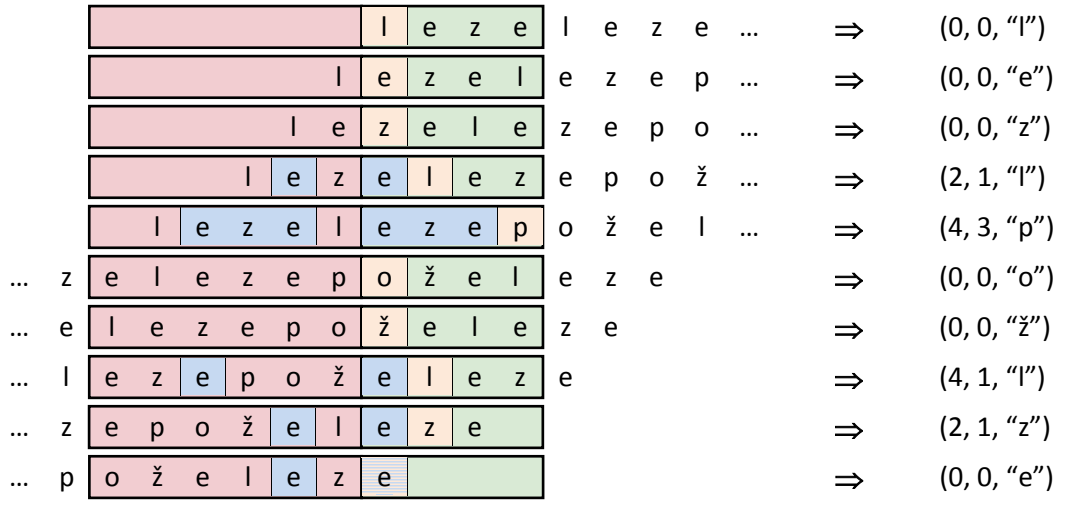
3.2 Komprese

Při provádění komprimace mohou nastat dva případy. V prvním případě se na konci textu vyskytuje znak, pro který neexistuje shoda, jak lze vidět na obrázku 4.



Obrázek 4 – Komprese LZ77 na konci bez shody („*lezelezepooceli*“)

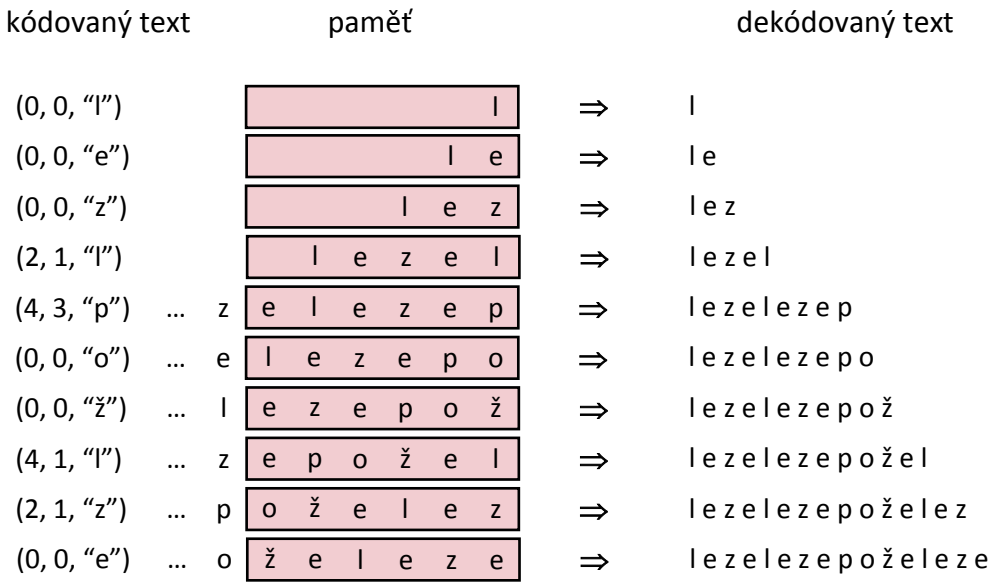
Ve druhém případě se na konci textu vyskytuje znak, který je obsažen v sekvenci shodných znaků nebo se jedná o shodu jediného znaku. Délku shody zmenšíme o jedna a parametr *následující znak* nabude hodnoty posledního znaku shody. Postup je znázorněn na obrázku 5.



Obrázek 5 - Komprese LZ77 na konci se shodou („*lezelezepoželeze*“)

3.3 Dekomprese

Za zmínku jistě stojí fakt, že dekomprese u algoritmu LZ77 je velmi jednoduchá a daleko méně náročná než komprese [9], viz obrázek 6. Dekompresní program si v paměti udržuje dekomprimovaný text v délce zakódované části. Čte trojice a z nich dekóduje opakující se části textu a znaky za nimi. Na dekompresi je podstatné, že je velmi rychlá. Zatímco komprese pro nalezení shody vyžaduje prohledávání zakódované části, dekomprese shodu pomocí pozice zjistí okamžitě [3].



Obrázek 6 – Dekomprese LZ77 (*„lezelezepoželeze“*)

4 LZSS

Mluvíme-li o algoritmu LZSS, jedná se v podstatě o efektivnější variantu kompresní metody LZ77. Řadí se mezi nejpoužívanější z modifikací algoritmu LZ77. Implementace této metody byla navržena Bellem v roce 1986 ([11]) podle idejí publikovaných J. Storerem a T. Szymanskim v roce 1982 ([12]).

4.1 Princip

Jednou z největších nevýhod algoritmu LZ77 je vytváření kódových trojic za každých okolností. V případě, že pro aktuálně kódovaný znak nebyla nalezena žádná shoda (v nezakódované části okénka se tento znak nenachází) nebo byla nalezena shoda pouze pro tento jediný znak, je na výstup poslána opět trojice. Velikost celé trojice je mnohem větší než velikost samotného znaku, který tato trojice popisuje.

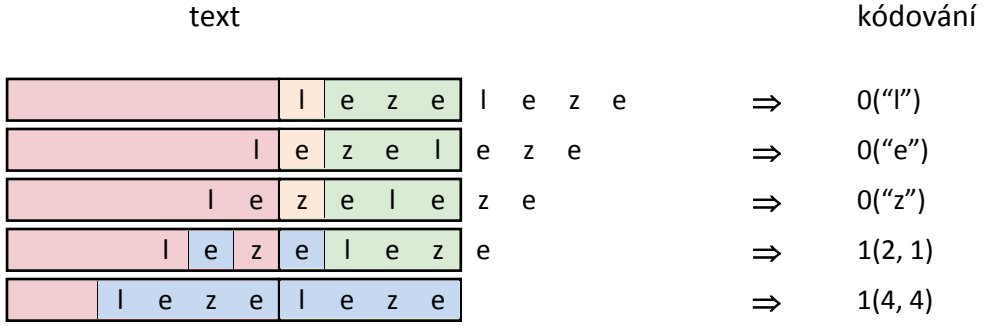
Pro výše zmiňovaný problém byla navržena právě metoda LZSS, jejíž výstupní data jsou tvořena *dvojicemi* parametrů nebo samostatnými *znaky*. Dvojice má tento tvar – *(pozice, délka)*, kde opět *pozice* vyjadřuje vzdálenost nalezené shody (shodných znaků) od konce zakódované části (někdy též označována jako *offset*) a *délka* vyjadřuje počet znaků nalezené shody. Obě čísla se ukládají jako čistý bitový zápis a tedy pro uložení *pozice* je potřeba $\log_2 N$ bitů, kde N je velikost okénka, a pro kódování délky je potřeba $\log_2 M$ bitů, kde M je maximální délka shody [2]. Znak ukládáme přímo jako osmibitové číslo [2]. Díky těmto poznatkům můžeme jednoduše určit, jestli je vhodné uložit dvojici nebo samostatný znak, a to pomocí vzorce

$$d = \left\lceil \frac{\log_2 N + \log_2 M}{8} \right\rceil,$$

kde **d** představuje minimální délku shody, pro kterou už je výhodné použít kódovou dvojici.

4.2 Komprese

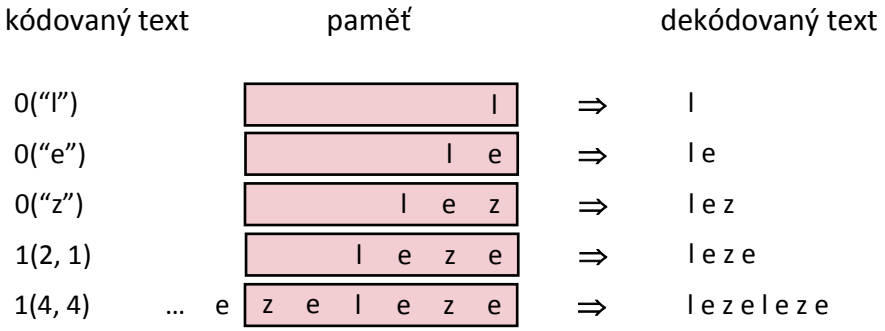
Celý postup je znázorněn na obrázku 7, příklad je uveden pro tyto hodnoty: N = 6, M = 4, d = 1.



Obrázek 7 – Komprese LZSS (*„lezeleze“)*

4.3 Dekomprese

Takzvaný pomocný bit, který je přidáván před dvojici nebo znak, slouží při dekompresi k určení toho, zda bude zpracovávána dvojice nebo znak. Jednotlivé kroky dekomprese lze vidět na obrázku 8.



Obrázek 8 – Dekomprese LZSS („*lezeleze*“)

5 Huffmanovo kódování

Huffmanovo kódování je pojmenované po svém objeviteli D. A. Huffmanovi, který jej popsal v roce 1952 ([10]). Zajímavostí je, že tento svůj objev si nikdy nenechal patentovat. Funguje na principu, že některý výskyt písmen v textu je častější než jiný. Pro znaky s častějším výskytem se používají kratší kódová slova a pro méně používané znaky delší kódová slova. Za zmínku stojí fakt, že tento princip použil již v roce 1800 Samuel Morse, kódoval tak písmena anglické abecedy a tím vznikla Morseova abeceda.

Toto kódování je ve své podstatě metoda na vytváření kódů s minimální redundancí, která vytváří kódovací strom – Huffmanův strom.

Nyní si popíšeme, jak toto kódování funguje:

1. projdi vstupní textový řetězec a vytvoř tabulku pravděpodobnosti výskytu (četností) jednotlivých znaků
2. vytvoř binární strom
    a. vytvoř seznam volných uzlů *w*, kde každý uzel je znak abecedy vstupního řetězce
    b. setřiď seznam *w* podle pravděpodobností znaků
    c. vytvoř nový uzel – rodič dvou uzlů (potomků) s nejmenší pravděpodobností ze seznamu *w*, pravděpodobnost nového uzlu je rovna součtu pravděpodobností obou potomků
    d. ze seznamu *w* odstraň oba potomky z kroku c.
    e. do seznamu *w* vlož rodiče z kroku c.
    f. setřiď seznam *w* podle pravděpodobností znaků
    g. opakuj od kroku c., dokud seznam *w* obsahuje více než jeden uzel
3. jednotlivým znakům přiřaď kódová slova
    a. projdi strom – všem levým hranám přiřaď hodnotu 0 a všem pravým hranám přiřaď hodnotu 1
    b. pro každý znak abecedy ulož kódové slovo, které je tvořeno hodnotami hran od kořene k uzlu s daným znakem

Tento druh kódování není příliš efektivní, protože délky kódů v bitech musí být zaokrouhleny na celá čísla. Přiřazení kódů je optimální pouze v případě, že je pravděpodobnost výskytu znaku mocninou $\frac{1}{2}$.

Statický model komprese dat předpokládá, že pravděpodobnosti znaků abecedy vstupního řetězce jsou dopředu známy, neboli kodér a dekodér používají stejný model. Naopak semiadaptivní model, jenž se přizpůsobuje vstupním datům (například Huffmanův strom), je nutné uchovat se zakódovanou zprávou pro pozdější dekompresi. Vhodnou metodou pro uchování Huffmanova stromu je uložení počtu znaků zdrojové abecedy, jejich výčet a rekurzivní výpis hran ohodnocení hran stromu, a to tak, že se nejprve vypisuje levý podstrom a poté se vypisuje pravý podstrom.

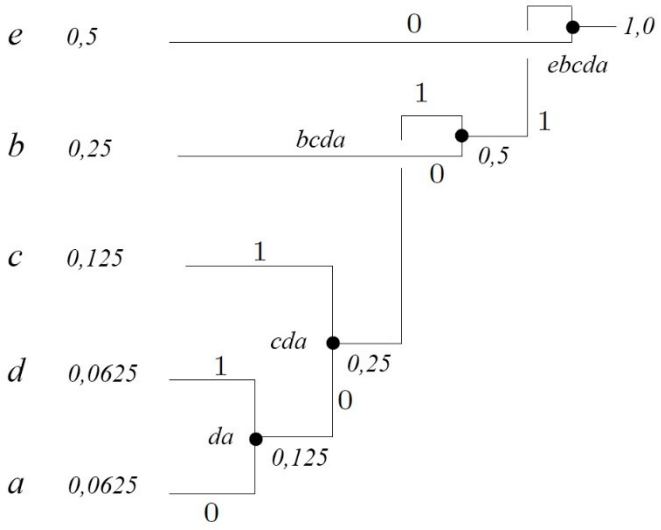
Dnes se s Huffmanovým kódováním setkáme v programech jako je PKZIP (Deflate), JPEG, MP3, BZIP2.

5.1 Příklad Huffmanova kódování:

- vstupní text: "***ebeeebcdeebbcaee***"

znak	**četnost výskytu**	**pravděpodobnost výskytů [%]**
e	8	0,5000
b	4	0,2500
c	2	0,1250
d	1	0,0625
a	1	0,0625

Tabulka 1 – Příklad Huffmanova kódování – tabulka četností



Obrázek 9 – Příklad Huffmanova kódování – Huffmanův strom

znak	četnost výskytu	Huffmanův kód
e	8	0
b	4	10
c	2	111
d	1	1101
a	1	1100

Tabulka 2 – Příklad Huffmanova kódování – výsledná kódová slova

Výsledný bitový zápis pro vstupní řetězec je 01000010111110100101011111 0000, jeho délka je tedy 30 bitů. Nesmíme však opomenout přičíst počet bitů potřebných k uložení vlastního Huffmanova stromu: *Fibonacci*(5){e,b,c,d,a}01010011 = 5+5*8+8 = 53 bitů (pojem *Fibonacci* zde označuje Fibonacciho kódování, viz kapitola 7). Celková velikost vstupních dat po kompresi je 30 + 53 = 83 bitů oproti původním 128 bitům. Výsledný kompresní poměr je $\frac{83}{128} \cong 0{,}65\ \%$.

5.2 Adaptivní Huffmanovo kódování

Adaptivní varianta Huffmanova kódování byla vyvinuta v roce 1973 N. Fallerem a nezávisle na něm v roce 1978 G. Gallagerem [13]. Vylepšená varianta Donalda E. Knutha z roku 1985, pojmenována FGC (Faller, Gallager, Knuth), je společně s Viterovým algoritmem jednou z nejznámějších variant adaptivního Huffmanova kódování.

Hlavním principem je předpoklad, že každý znak je alespoň jednou obsažen ve vstupním řetězci. Není nutné znát četnosti jednotlivých znaků před začátkem vytváření stromu. V průběhu kódování je strom přestavován a je aktualizován jeho obsah. Vzhledem k tom, že po dokončení kódování není potřeba uchovávat strukturu stromu, dochází ke zlepšení kompresního poměru. Tato metoda je jednoprůchodová.

5.3 Kanonické Huffmanovo kódování

Někdy je označováno za Huffman – Shannon – Fanovo kódování [14]. Zcela odpadá nutnost uchovávat binární strom. Jsou dána předem určená pravidla, podle kterých je binární kód pevně přiřazován znakům. K následné dekompresi je třeba znát použitá pravidla.

6 LZH a Deflate

6.1 LZH

Tuto metodu navrhl R. P. Brent v roce 1987 [2]. Její vylepšení oproti LZSS spočívá v tom, že pro kódování dvojic *(pozice, délka)* je použito Huffmanovo kódování, které je podrobně popsáno v kapitole 5.

Tato dvouprůchodová metoda využívá semiadaptivního Huffmanova kódování. Při prvním průchodu proběhne zpracování textu pomocí LZSS a zároveň se vypočtou statistické charakteristiky – pravděpodobnosti výskytu použitých znaků. Při druhém průchodu je již použito statické Huffmanovo kódování, které pracuje s vypočtenými pravděpodobnostmi. Zkomprimovaná data musí obsahovat i Huffmanův strom, který je potřebný pro dekompresi.

6.2 Deflate

Tento algoritmus popsal doc. Ing. Jan Platoš, Ph.D., ve své diplomové práci [2]:

> Jedná se o zajímavý způsob kódování výstupu použitý v algoritmu deflate programu ZIP. Jeho přesný popis naleznete v RFC1951. Tento algoritmus je upravený algoritmus LZSS, takže kóduje znaky a dvojice (offset, délka). Způsob kódování je navržen přímo pro konkrétní parametry, velikost okénka je stanovena na 32 kB, maximální shoda na 258 znaků a velikost abecedy na 256 symbolů.
>
> Vylepšením v této metodě je fakt, že nepotřebuje rozlišovací bity pro znak nebo kódovou dvojici. Řeší to tak, že sloučí znak i délku do jedné abecedy. Prvních 256 symbolů znamená znaky, kód 256 znamená konec bloku dat. Následuje 8 hodnot (257-264) reprezentujících přímo délky od 3 do 10. Kódy 265-284 reprezentují vždy skupinu délek a za daným kódem následuje několik bitů, pomocí kterých se rozliší, o jakou délku jde. Poslední kód 285 reprezentuje maximální možnou délku 258.
>
> V podobném duchu je realizováno i kódování offsetu. To je rozděleno do skupin pro 30 kódů. Kódy 0-3 reprezentují přímo vzdálenosti 1-4. Další kódy opět reprezentují skupinu délek a za vlastním kódem následuje několik bitů, které upřesní skutečnou hodnotu offsetu.
>
> Ukládání vlastních kódů je pak řešeno staticky pomocí prefixového kódu nebo pomocí Huffmanova kodéru.

6.2.1 Příklad deflate

Princip komprese si ukážeme na následujícím příkladu se vstupním řetězcem “***baaaabc***”:

1. první průchod

výsledek LZSS
(“b”)
(“a”)
(“a”)
(3,2)
(“b”)
(“c”)

znak	**pravděpodobnost výskytu**
a	$\frac{2}{7}$
b	$\frac{2}{7}$
c	$\frac{1}{7}$
2	$\frac{1}{7}$
3	$\frac{1}{7}$

Tabulka 3 – Deflate – výsledky prvního průchodu

2. druhý průchod (výsledek je třeba ještě doplnit o Huffmanův strom {5, {“c”, “b”, “3”, “2”, “a”, }, 00110011 }, viz kapitola 5)

výsledek komprese
01
11
11
100 101
01
00

znak	**Huffmanův kód**
a	11
b	01
c	00
2	110
3	010

Tabulka 4 – Deflate – výsledky druhého průchodu

7 Fibonacciho kódování

Fibonacciho kódy jsou prefixové kódy, pomocí kterých lze kódovat celá kladná čísla. Fibonacciho kód se je tvořen pomocí Fibonacciho čísel, která jsou definována rekurzivně:

$$F_1 = 1$$
$$F_2 = 1$$
$$F_i = F_{i-1} + F_{i-2}\,;\ \ i > 2$$

Posloupnost Fibonacciho čísel začíná čísly 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, … [16]

Vzorec pro kódování čísla *N*:

$$N = \sum_{i=0}^{k} d(i)\, F(i),$$

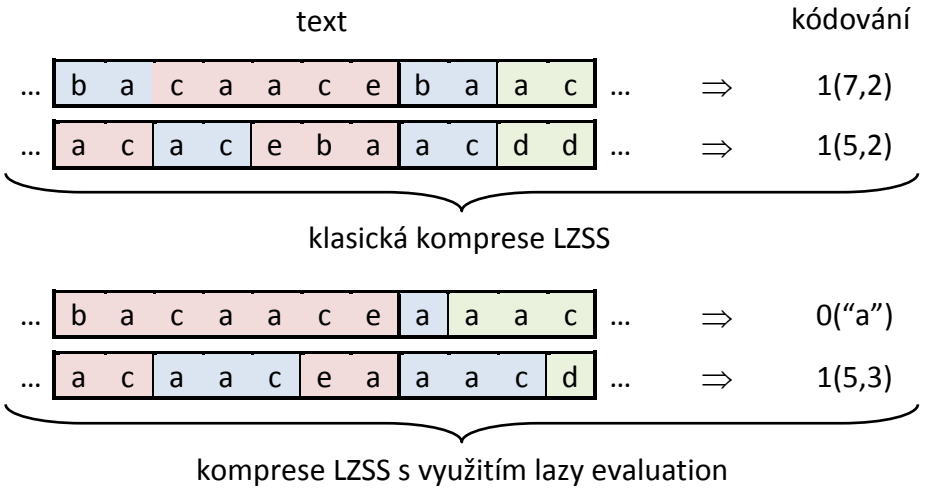
Dva následující koeficienty *d(i)* a *d(i+1)* nejsou nikdy nastaveny současně na jedna [17].

číslo	Fibonacciho kód	číslo	Fibonacciho kód
1	11	7	01011
2	011	8	000011
3	0011	9	100011
4	1011	10	010011
5	00011	11	001011
6	10011	12	101011

Tabulka 5 – Fibonacciho kódy

8 Lazy evaluation

Opožděné vyhodnocování (anglicky lazy evaluation či lazy matching) je metoda, kterou se dá dosáhnout velkého vylepšení kompresního poměru [2]. Z jiného hlediska je opožděné vyhodnocování programovací technikou, která odkládá vyhodnocení výrazu až na chvíli, kdy je toho zapotřebí. Tohoto tvrzení lze využít i u LZ77 a LZSS, a to tak, že kompresní algoritmus nezakóduje ihned první nalezenou shodu na pozici p, ale vyzkouší, zda na pozici $p + 1$ není nalezena delší shoda. V případě, že delší shoda nalezena není, pokračuje algoritmus obvyklým způsobem. V případě, že delší shoda nalezena je, postupuje algoritmus tak, že znak na pozici p je zakódován samostatně, shoda na pozici $p + 1$ zatím kódován není a opět se zkouší shoda na další pozici, tentokrát $p + 2$. Tento cyklus se opakuje, dokud nedosáhne předem určeného limitu – maximálního počtu pokusů, nebo dokud nachází lepší shodu. Vše lépe vysvětlí obrázek 10.



Obrázek 10 – Srovnání LZSS s a bez lazy evaluation

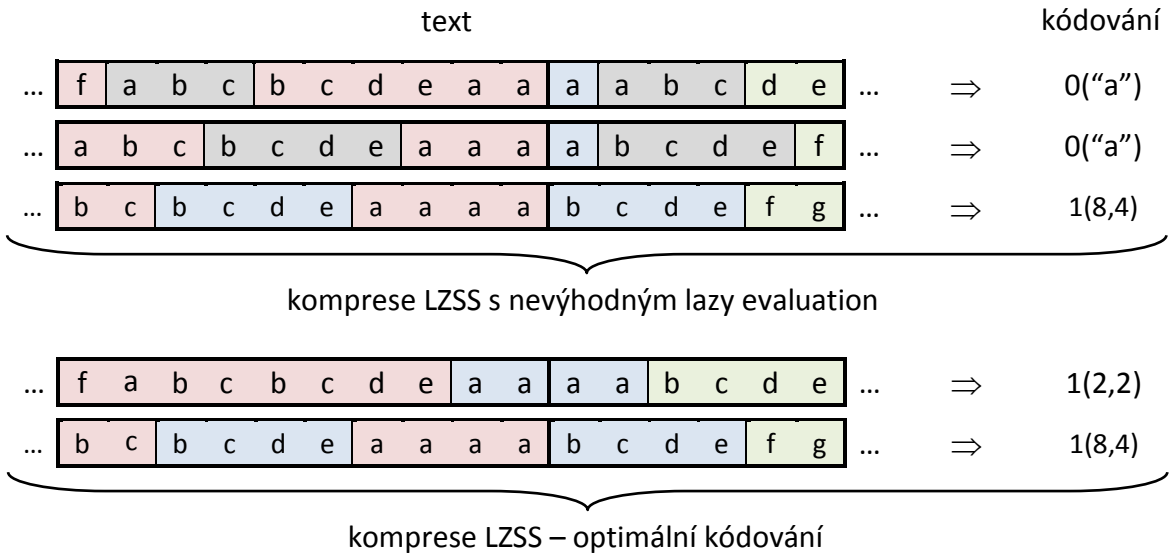
Převedeme-li příklad z obrázku 10 na počty bitů, pak dostaneme následující rovnice (celá nezáporná čísla kódujeme pomocí Fibonacciho kódování zohledňujícího i nulu a pro znak máme vyhrazeno 8 bitů):

$$\boldsymbol{LZSS} = Fibonacci(7) + Fibonacci(2) + Fibonacci(5) + Fibonacci(2) + 2\ bits$$
$$= 6 \text{ bits} + 4 \text{ bits} + 5 \text{ bits} + 4 \text{ bits} + 2 \text{ bits} = \mathbf{21\ bits}$$

$$\boldsymbol{LZSS\ s\ lazy\ evaluation} = 8\ bits + Fibonacci(5) + Fibonacci(3) + 2\ bits$$
$$= 8 \text{ bits} + 5 \text{ bits} + 4 \text{ bits} + 2 \text{ bits} = \mathbf{19\ bits}$$

Tedy na pouhých čtyřech znacích můžeme pozorovat rozdíl 2 bity ve prospěch kompresní metody využívající opožděného vyhodnocování.

Hlavní nevýhodou opožděného vyhodnocování je kódování samostatných po sobě jdoucích znaků i v případech, kdy lze tyto znaky zakódovat jedinou shodou (popřípadě více shodami), protože je na dalších pozicích opětovně nalezena delší shoda. Tento problém znázorňuje obrázek 111.



Obrázek 11 – Srovnání LZSS s nevýhodným lazy evaluation
a optimálního kódování (optimal coding)

Převedeme-li opět příklad z obrázku 11 na počty bitů, pak dostaneme následující rovnice (celá nezáporná čísla kódujeme pomocí Fibonacciho kódování zohledňujícího i nulu a pro znak máme vyhrazeno 8 bitů):

$$\textbf{\textit{LZSS s lazy evaluation}} = 8\ bits + 8\ bits + Fibonacci(8) + Fibonacci(4) + 3\ bits$$
$$= 8 \text{ bits} + 8 \text{ bits} + 6 \text{ bits} + 5 \text{ bits} + 3 \text{ bits} = \mathbf{30\ bits}$$

$$\textbf{\textit{LZSS optimal coding}}$$
$$= Fibonacci(2) + Fibonacci(2) + Fibonacci(8) + Fibonacci(4) + 2\ bits$$
$$= 4 \text{ bits} + 4 \text{ bits} + 6 \text{ bits} + 5 \text{ bits} + 2 \text{ bits} = \mathbf{21\ bits}$$

Zde se naopak opožděné vyhodnocování stává velmi nevýhodným oproti optimálnímu kódování, a to rozdílem 9 bitů při kódování šesti znaků.

9 Optimální parsování

Ve své bakalářské práci jsem se zabývala problematikou optimalizace lazy evaluation pro LZ77 (LZSS) s cílem vyvinout nový algoritmus pro zlepšení kompresního poměru, což se podařilo. Základem nového algoritmu byla myšlenka J. Storera a T. Szymanského s názvem optimální parsování (anglicky optimal parsing) ([15]), která je popsána níže.

Optimální parsování řeší případy, kdy kompresní algoritmus, v našem případě se zaměříme na LZSS, i s pomocí lazy evaluation, které výrazně zlepšuje kompresní poměr, na výstupu negeneruje nejlepší kódování. Tento problém byl popsán v předchozí kapitole. Slovní popis troj-průchodového algoritmu využívajícího optimální parsování k získání nejlepšího kódování je následující (pro zjednodušení si tento algoritmus pojmenujme *LZSS optimal*):

1. projdi celý vstupní text a pro každou pozici *x* doposud nezakódované části textu najdi nejdelší shodu *s*
2. ulož každou takovou shodu *s* (přesněji její délku) společně se znakem, pozicí nalezené shody a pomocnou proměnnou *r* pro uložení celkového počtu bitů (defaultně 0 bitů) do vhodné datové struktury, například pole *p*
3. projdi pole *p* odzadu dopředu a pro každou uloženou shodu *s* a celou podmnožinu kratších shod dané pozice *x* {*s*-1, *s*-2, …, 1, 0} (shoda délky nula zde představuje samotný znak, nikoliv dvojici čísel), vyhodnoť a ulož nejmenší počet bitů potřebných k uložení části textu od dané pozice po EOF, a to následovně:
    a. vypočti počet bitů potřebných k uložení dané shody *s* a její pozice nebo uložení samostatného znaku (celá nezáporná čísla kódujeme pomocí Fibonacciho kódování zohledňujícího i nulu a pro znak máme vyhrazeno 8 bitů)
    b. přičti počet bitů z předchozího kroku k pomocné proměnné *r* prvku *p*[*x*+*s*] (pro shodu *s*-1 přičítej k pomocné proměnné *r* prvku *p*[*x*+*s*-*1*] atd.)
    c. porovnej výsledné počty bitů z předchozího kroku pro množinu {*s*-1, *s*-2, …, 1, 0} a vyber shodu s nejkratším bitovým zápisem *s`*

d. změň v poli *p* původní délku shody *s* dané pozice *x* na shodu s nejkratším bitovým zápisem *s`* a ulož celkový počet bitů do pomocné proměnné *r* prvku *p*[*x*-*s`*]

4. projdi pole *p* a ulož výsledné kódové dvojice/znaky do výstupního souboru, posun v poli je prováděn vždy o délku shody *s`* aktuální pozice *x*

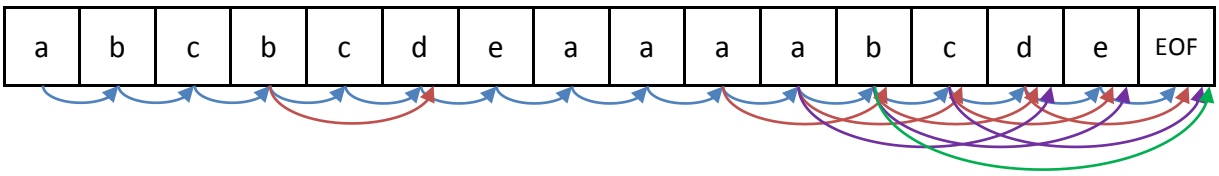
9.1 Příklad optimálního parsování:

Výše uvedený postup si ukážeme na příkladu pro konkrétní vstupní řetězec "***abcbcdeaaaabcde***":

1. vyhledání nejdelších shod pro všechny pozice ve vstupním řetězci – zapsáno do tabulky 6 a znázorněno na obrázku 12

pozice	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	EOF
délka shody	0	0	0	2	1	0	0	1	1	2	3	4	3	2	1	0

Tabulka 6 – Optimal parsing – nejdelší shody pro všechny pozice



Obrázek 12 – Optimal parsing – všechny varianty shod

2. uložení všech dat potřebných k následnému vyhodnocení nejkratšího bitového zápisu do tabulky 7

pozice	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**EOF**
pozice shody	0	0	0	2	2	0	0	7	1	2	10	8	8	8	8	0
délka shody	0	0	0	2	1	0	0	1	1	2	3	4	3	2	1	0
znak	a	b	c	b	c	d	e	a	a	a	a	b	c	d	e	EOF
celkový počet bitů	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabulka 7 – Optimal parsing – soupis dat

3. vyhodnocení nejkratšího bitového zápisu – do tabulky 8 ukládáme pro danou délku shody na dané pozici nejnižší počet bitů potřebný k uložení části textu od dané pozice po EOF

vstupní řetězec	a	b	c	b	c	d	e	a	a	a	a	b	c	d	e	EOF
délka shody **0**	**91**	**82**	**73**	72	64	**55**	**46**	**37**	30	30	**21**	20	20	18	9	**0**
1				71	**63**			38	**28**	29	22	21	21	19	**10**	
2				**64**						**21**	22	22	20	**11**		
3											22	21	**11**			
4												**12**				

Tabulka 8 – Optimal parsing – výsledné kódování

4. výsledné LZSS optimal kódování:

LZSS s využitím lazy evaluation	**výsledné LZSS optimal**
0(“a”)	0(“a”)
0(“b”)	0(“b”)
0(“c”)	0(“c”)
1(2,2)	1(2,2)
0(“d”)	0(“d”)
0(“e”)	0(“e”)
0(“a”)	0(“a”)
0(“a”)	1(1,1)
0(“a”)	1(2,2)
0(“a”)	1(8,4)
1(8,4)	

Tabulka 9 – Optimal parsing – srovnání LZSS s využitím lazy evaluation a výsledného LZSS optimal

$$\boldsymbol{LZSS\ s\ lazy\ evaluation}$$
$$= 8\ bits + 8\ bits + 8\ bits + Fibonacii(2)$$
$$+ Fibonacii(2) + 8\ bits + 8\ bits + 8\ bits + 8\ bits$$
$$+ 8\ bits + 8\ bits + Fibonacii(8) + Fibonacii(4)$$
$$+ 11\ bits$$
$$= 24\ bits + 4\ bits + 4\ bits + 48\ bits + 5\ bits + 3\ bits$$
$$+ 11\ bits = \mathbf{99\ bits}$$

$$\boldsymbol{LZSS\ optimal} = 8\ bits + 8\ bits + 8\ bits + Fibonacii(2)$$
$$+ Fibonacii(2) + 8\ bits + 8\ bits + 8\ bits$$
$$+ +Fibonacii(1) + Fibonacii(1) + Fibonacii(2)$$
$$+ Fibonacii(2) + Fibonacii(8) + Fibonacii(4)$$
$$+ 10\ bits$$
$$= 24\ bits + 4\ bits + 4\ bits + 24\ bits + 3\ bits + 4\ bits$$
$$+ 4\ bits + 6\ bits + 5\ bits + 3\ bits + 10\ bits = \mathbf{91\ bits}$$

Rozdíl mezi LZSS optimal a LZSS s využitím lazy evaluation činí 9 bitů při kompresi pouhých 15 znaků.

10 Zapojení Huffmanova kódování do LZSS s využitím lazy evaluation

Předchozí kapitoly byly mimo jiné věnovány principům lazy evaluation, metod LZH a Deflate. Algoritmus LZSS používá lazy evaluation ke zlepšení kompresního poměru prostřednictvím vyhodnocení výrazu až v případě potřeby. Metody LZH a Deflate zase vylepšují algoritmus LZSS pomocí Huffmanova kódování. Spojení těchto dvou myšlenek do jediného algoritmu sebou přináší velké výhody, především další zlepšení výsledného kompresního poměru, což je u komprese vždy velice žádoucí. Ovšem zapojení Huffmanova kódování do LZSS s využitím lazy evaluation, tedy zapojení ve smyslu provedení komprese LZSS s využitím lazy evaluation a následné použití statického Huffmanova kódování na výslednou sekvenci dvojic/znaků, s sebou nese i určité problémy, které je třeba vyřešit.

První problém je způsobený nevýhodou využití lazy evaluation u kompresních algoritmů (detailní popis v kapitole 8). Jedná se o ukládání jednotlivých znaků při nacházení delších shod na po sobě jdoucích pozicích, a to i v případě, že lze tyto znaky uložit prostřednictvím jedné nebo více shod. Tuto nevýhodu lazy evaluation lze vyřešit pomocí optimálního parsování, kterému je podrobně věnována kapitola 9.

Zde nastává druhý problém. U optimálního parsování dochází ke zpětnému průchodu vstupního textového řetězce, poté je pro všechny varianty nejdelší shody dané pozice vyhodnocen nejkratší bitový zápis potřebný k dosažení EOF, tím je dosaženo nejlepšího kódování. U komprese LZSS využívající lazy evaluation a statické Huffmanovo kódování však není předem znám přesný bitový zápis pro jednotlivé znaky, tyto informace jsou k dispozici až po provedení konečného Huffmanova kódování. Není tedy možné v průběhu komprese LZSS (s využitím lazy evaluation) vyhodnocovat nejkratší bitový zápis dané pozice potřebný k dosažení EOF, neboť k tomu nemáme potřebné informace.

Výše popsané problémy řeší optimalizovaný algoritmus LZSS využívající lazy evaluation, Huffmanovo kódování a optimální parsování k získání nejlepšího kódování. U tohoto optimalizovaného algoritmu je tabulka četností, potřebná k provedení Huffmanova kódování, společná pro délky shod a znaky. Velikost tabulky a její obsah se aktualizuje v průběhu algoritmu, maximální velikost tabulky by však neměla nepřekročit velikost ASCII tabulky, tedy 256. Proto je více než vhodné omezit maximální délku shody také na 256. Pro uložení pozice využijeme Fibonacciho kódování. Jeho slovní popis je následující (pro zjednodušení si tento algoritmus pojmenujme *LZSS Huffman optimal*):

1. projdi celý vstupní text a pro každou pozici x doposud nezakódované části textu najdi nejdelší shodu s
2. ulož každou takovou shodu s (přesněji její délku) společně se znakem a pozicí nalezené shody do vhodné datové struktury, například pole p

3. projdi pole *p* – otestuj aktuální pozici *x* a pozici *x+1*, zda lze využít lazy evaluation:

3.1. v negativním případě ulož kódovou dvojici/znak pro aktuální pozici *x* a aktualizuj tabulku četností

3.2. v pozitivním případě proveď:

a. od aktuální pozice *x* – všechny (mimo poslední) po sobě jdoucí nejdelší shody, které splňují pravidlo, že každá shoda *t* na pozici *y* je vždy delší než shoda na pozici *y+1*, ulož do pomocného pole *pp* znak, délku shody, pozicí shody a pomocnou proměnnou *r* pro uložení celkového počtu bitů (defaultně 0 bitů)

b. projdi pole *pp* odzadu dopředu a pro každou uloženou shodu *t* a celou podmnožinu kratších shod dané pozice *y* {*t-1, t-2, ..., 1, 0*} (shoda délky nula zde představuje samotný znak, nikoliv dvojici čísel), vyhodnoť a ulož nejmenší počet bitů potřebných k uložení části textu od dané pozice po EOF, a to následovně:

b.1.1. dvojice *(pozice, délka)* – pro pozici nalezené shody *s* vypočti počet bitů prostřednictvím Fibonacciho kódování a pro délku nalezené shody *s* dočasně aktualizuj tabulku četností a vypočti počet bitů pomocí Huffmanova kódování a dočasně aktualizované tabulky četností

b.1.2. samostatný *znak* - dočasně aktualizuj tabulku četností a vypočti počet bitů pomocí Huffmanova kódování a dočasně aktualizované tabulky četností

→ v krocích b.1.1. a b.1.2. dochází k časté dočasné aktualizaci tabulky četností a sestavování Huffmanova stromu, zde je možné využít aproximace Huffmanova kódování (problematika je popsána v podkapitole 10.2.) – v našem případě je ale žádoucí přesnější výpočet bitů, tedy tato aproximace nebude využita

b.2. vrať dočasně aktualizovanou tabulku četností do původního stavu

b.3. přičti počet bitů z kroku b.1.1. nebo b.1.2. k pomocné proměnné *r* prvku *p*[*y*+*t*] (pro shodu *t-1* přičítej k pomocné proměnné *r* prvku *p*[*y*+*t-1*] atd.)

b.4. porovnej výsledné počty bitů z předchozího kroku pro množinu {*t-1, t-2, ..., 1, 0*} a vyber shodu s nejkratším bitovým zápisem *s`*

b.5. změň v poli *pp* původní délku shody *t* dané pozice *y* na shodu s nejkratším bitovým zápisem *t`* a ulož celkový počet bitů do pomocné proměnné *r* prvku *p*[*y*-*t`*]

c. projdi pole *pp* a ulož výsledné kódové dvojice/znaky do výstupního souboru, posun v poli je prováděn vždy o délku shody *t`* aktuální pozice *y*

d. není-li dosaženo EOF, posuň se v poli *p* o délku pole *pp* - 1 a proveď krok 3

4. není-li dosaženo EOF, posuň se v poli *p* o délku shody *s* aktuální pozice *x* a proveď krok 3

5. pomocí Huffmanova kódování nad tabulkou četností ulož výslednou sekvenci dvojic/znaků do výstupního souboru

10.1 Příklad LZSS Huffman optimal:

Výše uvedený postup si ukážeme na příkladu pro konkrétní vstupní řetězec "***abcaaaabc***":

1. vyhledání nejdelších shod pro všechny pozice ve vstupním řetězci – zapsáno do tabulky 10

pozice	1	2	3	4	5	6	7	8	9	EOF
délka shody	0	0	0	1	1	2	3	2	1	0

Tabulka 10 – LZSS Huffman optimal – nejdelší shody pro všechny pozice

2. uložení všech dat potřebných k následnému vyhodnocení nejkratšího bitového zápisu do tabulky 11

pozice	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**EOF**
pozice shody	0	0	0	3	1	2	6	6	6	0
délka shody	0	0	0	1	1	2	3	2	1	0
znak	a	b	c	a	a	a	a	b	c	EOF
celkový počet bitů	0	0	0	0	0	0	0	0	0	0

Tabulka 11 – LZSS Huffman optimal – soupis dat

3. pro první čtyři pozice nelze využít lazy evaluation, výsledkem prvních čtyř iterací je sekvence {0(“a”), 0(“b”), 0(“c”), 0(“a”)} a tabulka četností, viz tabulka 12

znak	**četnost výskytu**
a	2
b	1
c	1

Tabulka 12 – LZSS Huffman optimal – tabulka četností

- u pozic 5 a 6 lze využít lazy evaluation, vyznačeno v tabulce 6, do tabulky 13 ukládáme pro danou délku shody na dané pozici nejnižší počet bitů potřebný k uložení části textu od dané pozice po EOF (v tomto případě EOF znamená konec lazy evaluation), tyto nejnižší počty bitů získáme pomocí Huffmanova kódování

vstupní řetězec		a	a	EOF
délka shody	0	4	2	0
	1	7	8	
	2		8	

Tabulka 13 – LZSS Huffman optimal – výsledné kódování případu využití lazy evaluation

- výsledkem tabulky 13 je sekvence {0(“a”), 0(“a”)}

- pro další tři pozice opět nelze využít lazy evaluation, výsledkem je dvojice {1(6,3)}

5. výsledné kódování LZSS Huffman optimal (v tomto případě je výsledná sekvence dvojic/znaků totožná s LZSS s využitím lazy evaluation):

LZSS s využitím lazy evaluation	výsledné LZSS Huffman optimal
0(“a”)	0(“a”)
0(“b”)	0(“b”)
0(“c”)	0(“c”)
0(“a”)	0(“a”)
0(“a”)	0(“a”)
0(“a”)	0(“a”)
1(6,3)	1(6,3)

Tabulka 14 – LZSS Huffman optimal – srovnání LZSS s využitím lazy evaluation a výsledného LZSS Huffman optimal

znak	četnost výskytu	Huffmanův kód
a	4	1
b	1	00
c	1	011
3	1	010

Tabulka 15 – LZSS Huffman optimal – tabulka četnosti a Huffmanova kódování

- k výslednému kódování je třeba ještě doplnit Huffmanův strom {4, {“b”, “3”, “c”, “a”, }, 001011 }, viz kapitola 5)

$$\boldsymbol{LZSS\ s\ lazy\ evaluation}$$
$$= 6 * 8\ bits + Fibonacii(6) + Fibonacii(3) + 7bits$$
$$= 48\ bits + 5\ bits + 4\ bits + 7\ bits = \mathbf{64\ bits}$$

$$\boldsymbol{LZSS\ Huffman\ optimal}$$
$$= 4 * 1\ bits + 2\ bits + 3\ bits + 3\ bit + Fibonacii(6)$$
$$+ 7\ bits + Fibonacci(4) + 3 * 8\ bits + 6\ bits$$
$$= 4\ bits + 8\ bits + 5\ bits + 7\ bits + 5\ bits + 24\ bits$$
$$+ 6\ bits = \mathbf{59\ bits}$$

Rozdíl mezi kódováním a LZSS Huffman optimal a LZSS s využitím lazy evaluation činí 5 bitů při kompresi 9 znaků.

10.2 Možnosti aproximace statistického kódování

Statistické kódování vytváří pro daný znak abecedy kódové slovo proměnné délky, která se odvíjí od pravděpodobnosti výskytu znaku (četnosti znaku) v komprimovaném textu. Pomocí těchto pravděpodobností je sestaven model (například Huffmanův strom), dle kterého je sestavováno kódové slovo. V případě potřeby je přepočítávání (aktualizace) modelů často velmi nákladná, proto je vhodné využít nějaké aproximace. Aproximace je v podstatě odhad, nejčastěji dolní odhad, velikosti kódového slova pro daný znak v bitech.

U Huffmanova kódování lze pro aproximaci využít následující vzorec:

$$H(x) = -log_2 p(x),$$

kde *x* je znak abecedy, *p(x)* je pravděpodobnost výskytu znaku *x*. Další možností je sestavovat model jednou za několik kroků (iterací).

U aproximace Shannon-Fanova kódování jde o stejný postup jako v případě Huffmanova kódování (oba algoritmy jsou téměř totožné, u Shannon-Fanova kódování je strom tvořen opačně, tedy od kořene k listům).

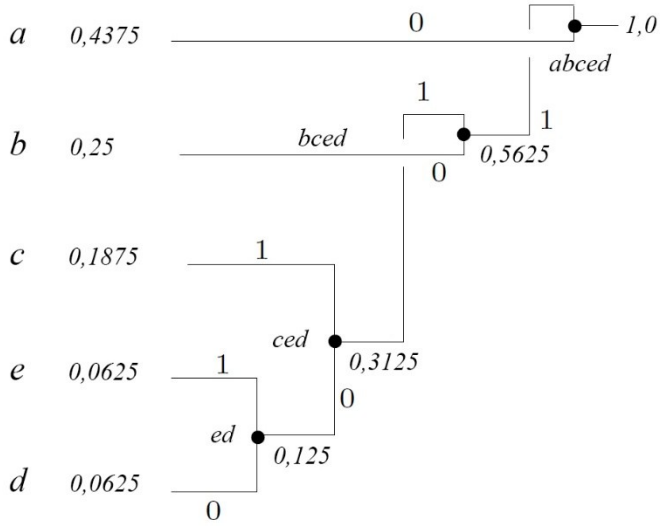
Aproximace u aritmetického kódování se používá zejména k urychlení vlastního kódování, což je velmi náročná operace.

10.2.1 Příklad aproximace Huffmanova kódování:

- vstupní text: "***abcaabcdaabbceaa***"

znak	četnost výskytu	pravděpodobnost výskytů [%]
a	7	0,4375
b	4	0,2500
c	3	0,1875
e	1	0,0625
d	1	0,0625

Tabulka 16 – Příklad aproximace Huffmanova kódování – tabulka četností



Obrázek 13 – Příklad aproximace Huffmanova kódování – Huffmanův strom

znak	Huffmanův kód	velikost kódu [bit]	odhad velikosti $-log_2 p(x)$ [bit]
a	0	1	1,19
b	10	2	2,00
c	111	3	2,42
e	1101	4	4,00
d	1100	4	4,00

Tabulka 17 – Příklad aproximace Huffmanova kódování – Huffmanův kód

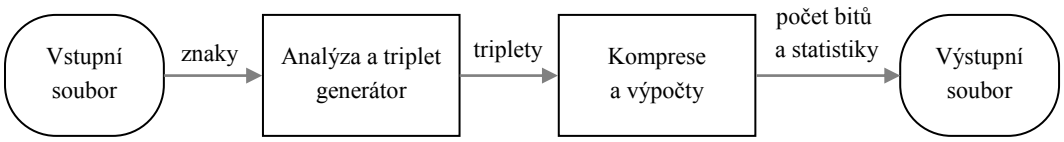
11 Implementace

Pro implementaci experimentálního programu byl zvolen jazyk C# a prostředí Visual Studio 2013, .NET Framework 4.5.

Před vlastním popisem implementace je třeba zdůraznit, že přesto, že nad daným vstupním textovým souborem probíhá komprese, výstupem programu není bitový zápis komprese, ale jeho velikost, tedy celkový počet bitů potřebných k uložení tohoto bitového zápisu, a další statistická data, včetně výsledků srovnávacích kompresí.

Program probíhá ve dvou fázích:

1. Analýza vstupního souboru a tvorba tripletů
2. Vlastní komprese, výpočet velikosti bitového zápisu a statistických dat



Obrázek 14 – Implementace

V první fázi se prochází celý soubor a pro pozici každého vstupního znaku se generuje triplet (*pozice maximální nalezené shody, délka maximální nalezené shody, aktuální znak*), výsledkem první fáze je tedy sekvence tripletů.

Ve druhé fázi je pak použita kompresní metoda LZSS s lazy evaluation. Každý triplet je otestován na případ využití lazy evaluation, v negativním případě je výsledkem komprese daného tripletu standardní dvojice (pozice, délka) nebo znak a aktualizace tabulky četností pro pozdější Huffmanovo kódování. V kladném případě se využije optimal parsing a vyhodnocení všech variant maximálních shod pomocí okamžitého výpočtu Huffmanova stromu pro aktualizovanou tabulku četností. Poté je sestaven Huffmanův strom pro finální tabulku četností a vypočítá se velikost bitového zápisu pro uložení zkomprimovaných dat. Tento výsledný počet bitů a další statistická data (počet znaků, velikost okénka, počty délek shod souboru, počty délek po sobě jdoucích případů lazy evaluation, velikosti bitových zápisů dalších kompresních metod) se ukládají do textového souboru.

Použitá implementace kompresních metod je zcela jednoduchá, protože cílem práce nebylo nalézt efektivní implementace těchto metod, ale zefektivnit kompresi LZSS s využitím lazy evaluation, Huffmanova kódování a jeho aproximace ve smyslu zlepšení kompresního poměru.

Implementace Fibonacciho kódování, popsaného v kapitole 7, rozšiřuje toto kódování o schopnost kódovat i nulu, a to tak, že ke každé kódované hodnotě je přičtena jednička. Při dekompresi je nutné tuto jedničku opět odečíst.

Jádrem programu je třída *AproximationSCforImprovementLE*, v jejíž metodě *TestingImprovement()* jsou volány metody:

- *LZSSLEimproveHuff()* – implementace optimalizovaného LZSS využívajícího lazy evaluation a Huffmanovo kódování (*LZSS Huffman optimal*)
- *LZSSLEimprove()* – implementace optimalizovaného LZSS využívajícího lazy evaluation (*LZSS optimal*)
- *LZSSLE()* – implementace LZSS využívajícího lazy evaluation a LZSS využívajícího lazy evaluation i Huffmanovo kódování
- *LZSS* – implementace LZSS a LZSS využívajícího Huffmanovo kódování

12 Testování

Po teoretické části a kapitolách popisujících problematiku zapojení Huffmanova kódování do LZSS s využitím lazy evaluation následuje fáze testování. Cílem tohoto testování je ověřit efektivitu navrženého algoritmu *LZSS Huffman optimal*. Testování bude probíhat nad testovacími soubory z tzv. The Canterbury Corpus a The Calgary Corpus[7].

The Calgary Corpus vnikl na konci 80. let 20. století a v průběhu 10 let se stal de facto standardem pro testování efektivity a rychlosti bezztrátových kompresních algoritmů[7]. Dnes už je poněkud neaktuální. Je tvořen 14 soubory různého charakteru.

The Canterbury Corpus byl sestaven v roce 1997 z typických příkladů běžně používaných dat za účelem nahrazení kolekce The Calgary Corpus. Obsahuje 11 souborů, z různých zdrojů.

Pro toto testování je použit výběr z obou kolekcí The Calgary Corpus (tabulka 18) i The Canterbury Corpus (tabulka 19).

Soubor	Popis	Velikost (v bytech)
bib	textová databáze knih	111 261
book1	kniha	768 771
book2	kniha (troff formát)	610 856
news	komunikace USENETu	377 109
paper1	technický dokument	53 161
paper2	technický dokument	82 199
paper3	technický dokument	46 526
paper4	technický dokument	13 286
paper5	technický dokument	11 954
paper6	technický dokument	38 105
progc	kód C	39 611
progl	kód LISP	71 646
progp	kód PASCAL	49 379
trans	přepis terminálové komunikace	93 695

Tabulka 18 – Testovací soubory z kolekce The Calgary Corpus

Soubor	Popis	Velikost (v bytech)
alice29.txt	anglický text	152 089
asyoulik.txt	scénář hry	125 179
cp.html	kód HTML	24 603
fields.c	kód jazyka C	11 150
grammar.lsp	kód LISP	3 721
lcet10.txt	technický dokument	426 754
plrabn12.txt	knižní dokument	481 861
xargs.1	GNU manuálové stránky	4 227

Tabulka 19 – Testovací soubory z kolekce The Canterbury Corpus

Parametry testování:

- testovací soubory – tabulky 18 a 29
- velikost posuvného okénka – 512 B, 1 kB, 2 kB, 4 kB, 8 kB, 16 kB, 32 kB
- délka maximální shody – pro všechny testy bude použita pevná délka 256 B

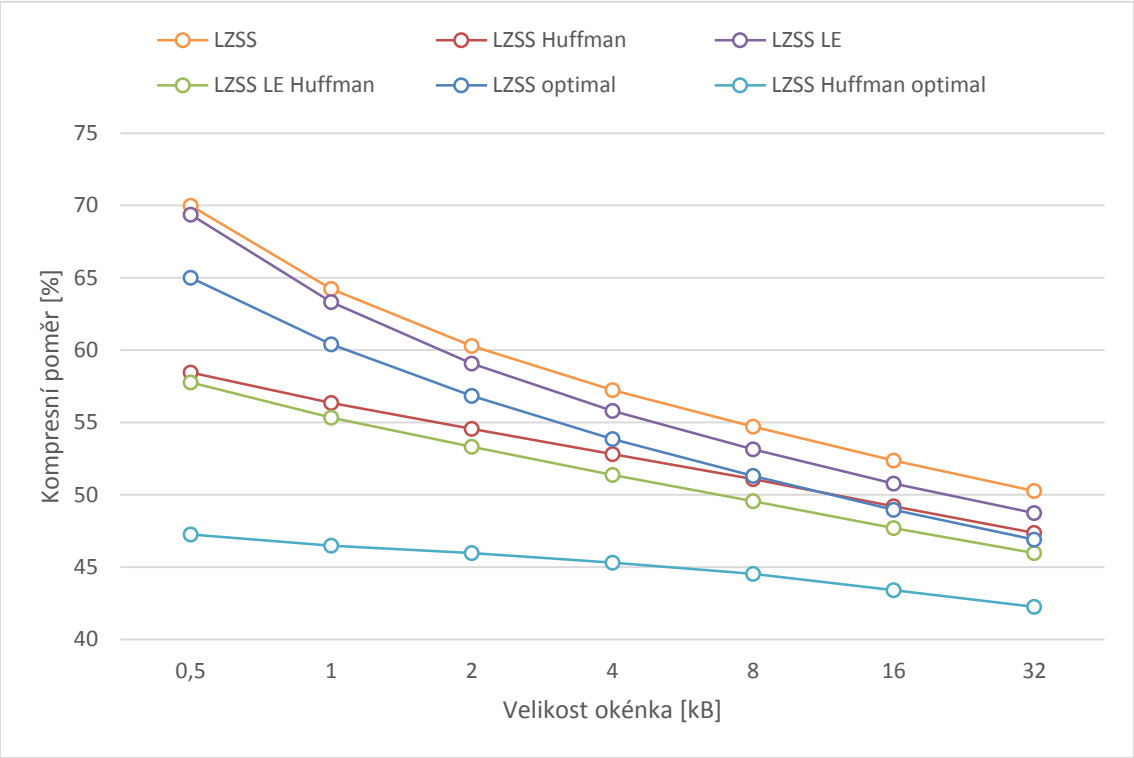
Zkratky použitých kompresních algoritmů:

- ***LZSS*** – klasický algoritmus LZSS, ukládá dvojice (*pozice, délka*) a *znaky*
- ***LZSS LE*** – LZSS vylepšený o lazy evaluation
- ***LZSS Huffman*** – LZSS využívající Huffmanovo kódování
- ***LZSS LE Huffman*** – LZSS využívající lazy evaluation a Huffmanovo kódování
- ***LZSS optimal*** – optimalizované LZSS pospané v kapitole 9
- ***LZSS Huffman optimal*** – optimalizované LZSS využívající Huffmanovo kódování pospané v kapitole 10

U algoritmů *LZSS, LZSS LE, LZSS optimal* je pro *znak* vymezeno 8 bitů a pro dvojice *pozice, délka* je použito Fibonacciho kódování.

U algoritmů *LZSS Huffman, LZSS LE Huffman, LZSS Huffman optimal* je pro *znak* a *délku* použito Huffmanovo kódování, pro *pozici* je opět použito Fibonacciho kódování.

12.1 Testování souboru alice29.txt



Obrázek 15 – Graf výsledků komprese souboru alice29.txt

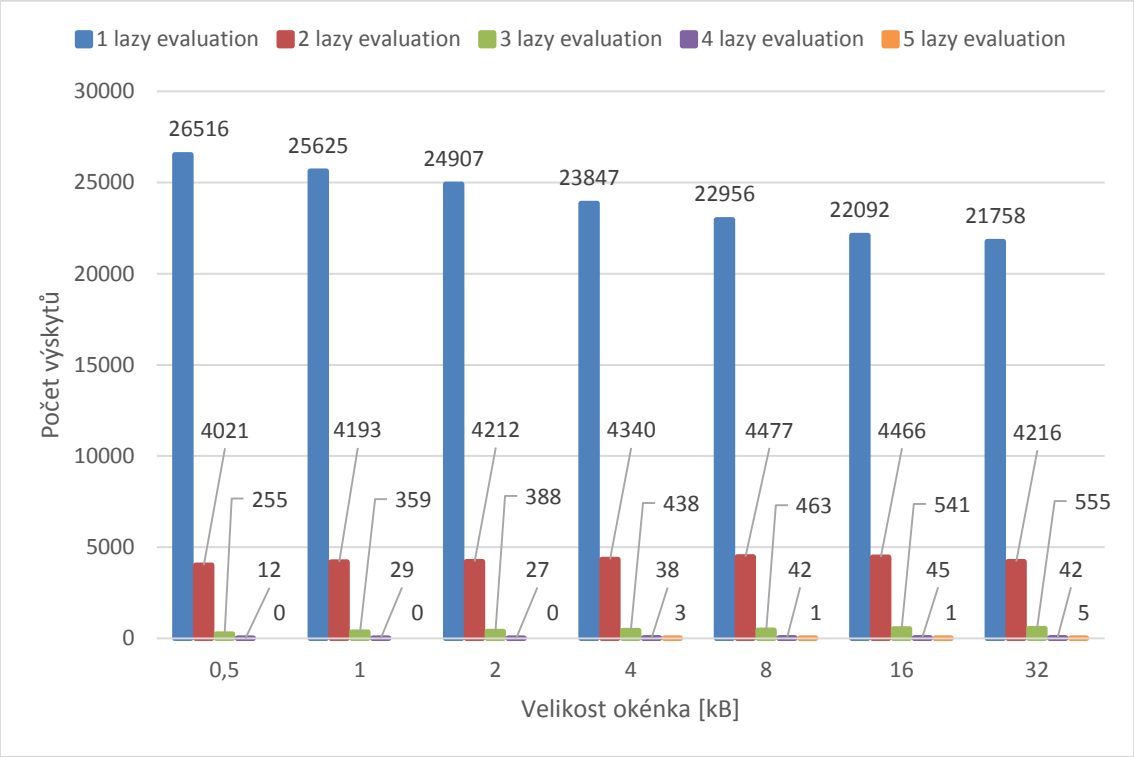
Z grafu na obrázku 15 je patrné, že se kompresní poměr vybraných algoritmů zmenšuje společně s rostoucí velikosti okénka, tudíž nejlepších výsledků (nejmenších kompresních poměrů) bylo dosaženo pro okénko o velikosti 32 kB. Dle očekávání se jako nejúčinnější kompresní algoritmus ukázal nově navržený LZSS Huffman optimal. V porovnáním s ostatními algoritmy je u LZSS Huffman mírnější průběh, rozdíl mezi kompresním poměrem pro velikost okénka 0,5 kB a 32 kB je 5,01 %, což při původní velikosti souborů 152 089 bytů činí 7 627 bytů. Pro okénko 16 kB a 32 kB dosahuje LZSS optimal lepších výsledků než LZ Huffman o téměř 0,5%. Výsledky LZSS potvrzují, že se jedná o nejméně efektivní algoritmus z dané šestice. Podrobné údaje o výsledcích všech použitých kompresní jsou uvedeny v tabulce 20.

Velikost okénka [kB]	LZSS		LZSS Huffman		LZSS LE	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	106 451	69,99	88 896	58,45	105 504	69,37
1	97 694	64,23	85 727	56,37	96 312	63,33
2	91 712	60,30	82 979	54,56	89 856	59,08
4	87 055	57,24	80 328	52,82	84 877	55,81
8	83 233	54,73	77 703	51,09	80 836	53,15
16	79 639	52,36	74 856	49,22	77 220	50,77
32	76 472	50,28	72 056	47,38	74 119	48,73

Velikost okénka [kB]	LZSS LE Huffman		LZSS optimal		LZSS Huffman optimal	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	87 863	57,77	98 880	65,01	**71 895**	**47,27**
1	84 155	55,33	91 872	60,41	**70 687**	**46,48**
2	81 099	53,32	86 464	56,85	**69 939**	**45,99**
4	78 153	51,39	81 897	53,85	**68 917**	**45,31**
8	75 394	49,57	78 048	51,32	**67 752**	**44,55**
16	72 569	47,71	74 485	48,97	**66 018**	**43,41**
32	69 937	45,98	71 347	46,91	**64 268**	**42,26**

Tabulka 20 – Výsledky kompresí souboru alice29.txt
(V – velikost zakódovaného textu v bytech, KP – kompresní poměr v %,
nejlepší výsledky pro jednotlivá okénka jsou zvýrazněny tučně)

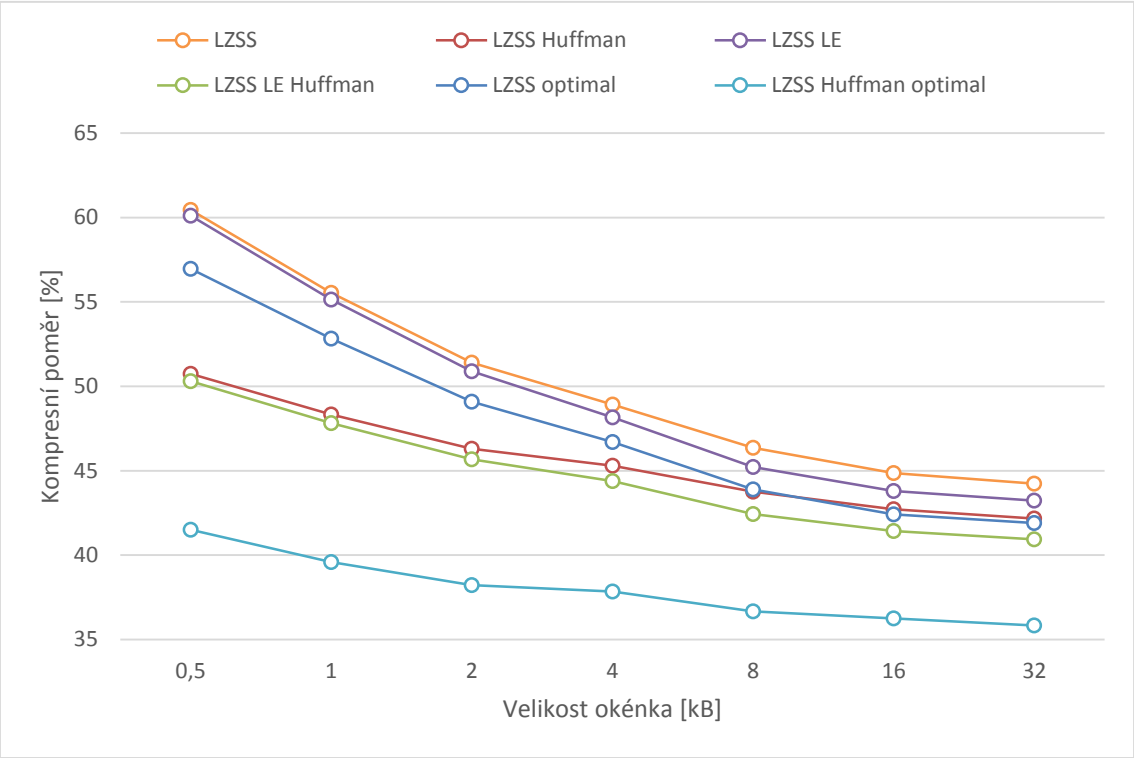
12.1.1 Srovnání výskytu po sobě jdoucích lazy evaluation



Obrázek 16 – Graf výskytu lazy evaluation u souboru alice29.txt

Graf na obrázku 16 popisuje četnost výskytu po sobě jdoucích opožděných vyhodnocování v závislosti na velikosti okénka – například *3 lazy evaluation* tedy znamená tři po sobě jdoucí případy opožděného vyhodnocování při kompresi. Pro soubor alice29.txt je nejvíce případů po sobě jdoucích lazy evaluation v délce 1, nejméně v délce 4 pro okénka 0,5 kB – 2kB a v délce 5 pro okénka 4 kB – 32 kB. Z grafu vyplývá, jak markantní rozdíl je mezi nejčastějším a nejméně častým výskytem.

12.2 Testování souboru cp.html



Obrázek 17 – Graf výsledků komprese souboru cp.html

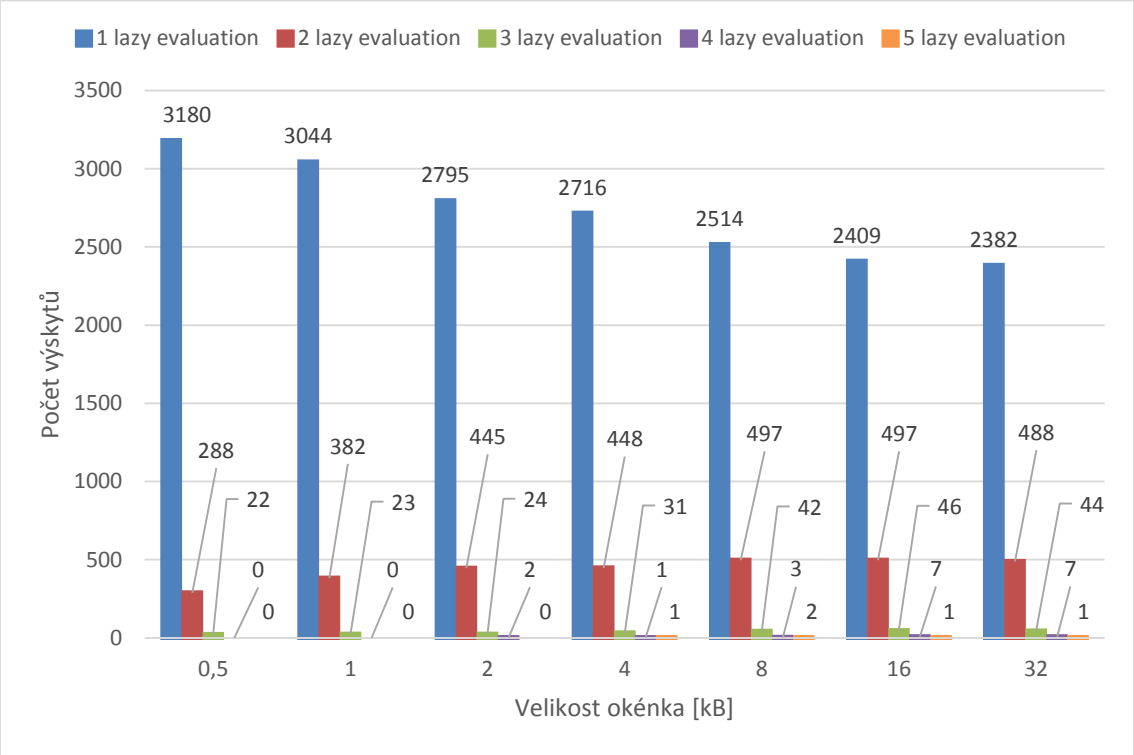
Graf na obrázku 17 zobrazuje průběh kompresí pro soubor s příponou html. Algoritmus LZSS Huffman optimal si zachovává svou efektivitu vzhledem k rostoucí velikosti okénka a dosahuje kompresního poměru 35,8 % – 41,5 %. Všechny hodnoty výsledků pro tento soubor jsou zaneseny do tabulky 21.

Velikost okénka [kB]	LZSS		LZSS Huffman		LZSS LE	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	14 873	60,45	12 484	50,74	14 788	60,11
1	13 664	55,54	11 892	48,34	13 567	55,14
2	12 646	51,40	11 391	46,30	12 520	50,89
4	12 034	48,91	11 146	45,30	11 851	48,17
8	11 405	46,36	10 766	43,76	11 127	45,22
16	11 038	44,86	10 509	42,71	10 774	43,79
32	10 883	44,23	10 372	42,16	10 636	43,23

Velikost okénka [kB]	LZSS LE Huffman		LZSS optimal		LZSS Huffman optimal	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	12 376	50,30	14 013	56,95	**10 211**	**41,50**
1	11 763	47,81	12 998	52,83	**9 740**	**39,59**
2	11 237	45,67	12 078	49,09	**9 404**	**38,22**
4	10 922	44,39	11 490	46,70	**9 312**	**37,85**
8	10 438	42,42	10 800	43,90	**9 022**	**36,67**
16	10 193	41,43	10 435	42,41	**8 916**	**36,24**
32	10 073	40,94	10 308	41,90	**8 816**	**35,83**

Tabulka 21 – Výsledky kompresí souboru cp.html
(V – velikost zakódovaného textu v bytech, KP – kompresní poměr v %,
nejlepší výsledky pro jednotlivá okénka jsou zvýrazněny tučně)

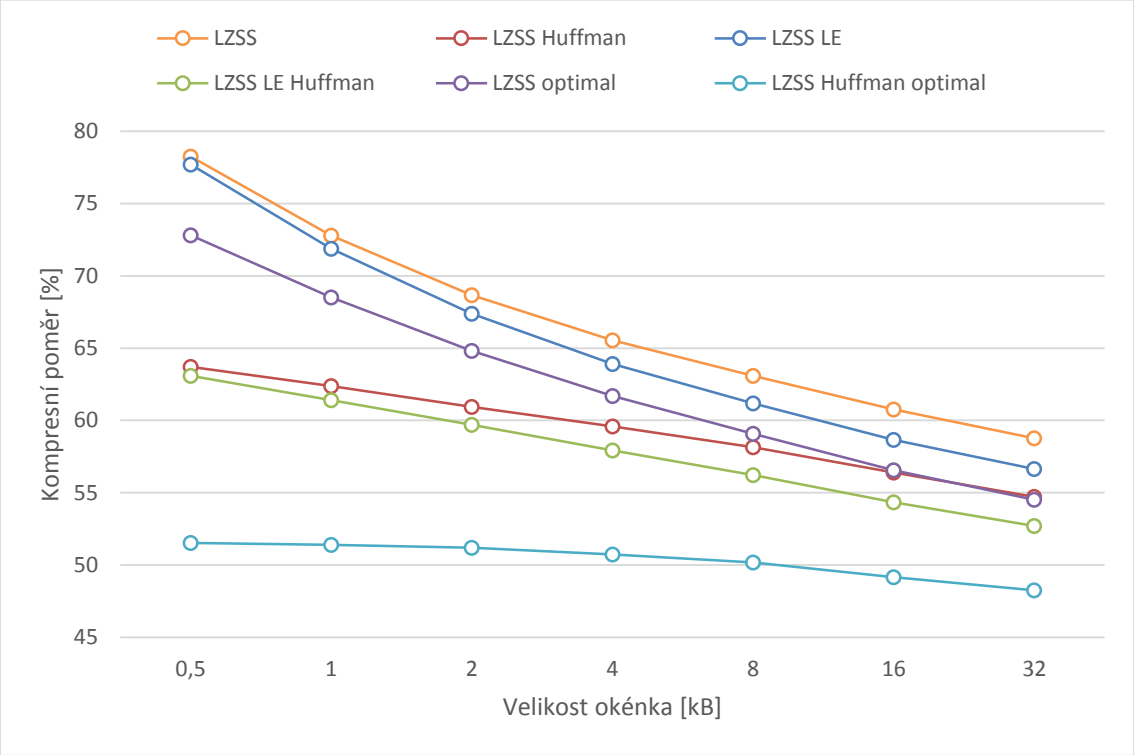
12.2.1 Srovnání výskytu po sobě jdoucích lazy evaluation



Obrázek 18 – Graf výsledků komprese souboru cp.html

U grafu (obrázek 18), který srovnává počet výskytů po sobě jdoucích případu lazy evaluation pro soubor cp.html, se opakuje trend jako u souboru alice29.txt. S rostoucí velikostí okénka klesá počet krátkých výskytů a roste počet dlouhých výskytu lazy evaluation

12.3 Testování souboru book1.txt



Obrázek 19 – Graf výsledků komprese souboru book1.txt

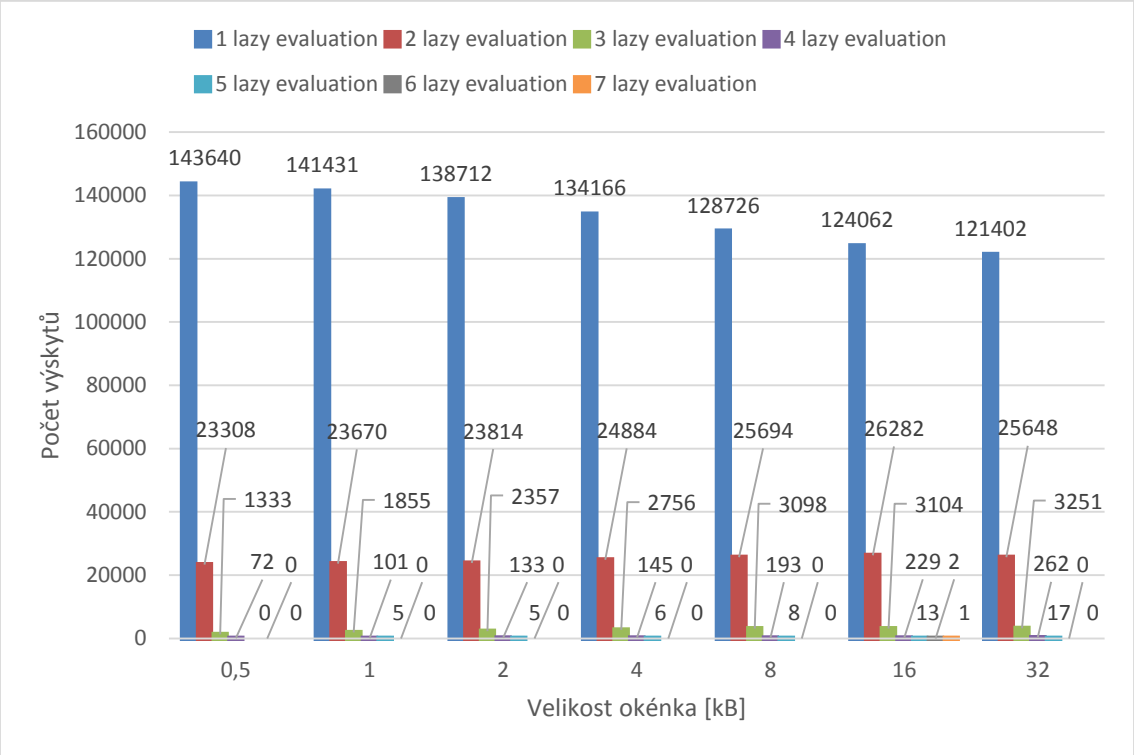
Z grafu výsledků souboru book1 lze vyčíst, že v tomto případě rostla efektivita navrženého algoritmu LZSS Huffman optimal (v závislosti na velikosti okénka) jen velmi pozvolna. U různorodý anglického textu jako je v book1.txt nikdy nedosáhneme takového kompresního poměru jako u cp.html

Velikost okénka [kB]	LZSS		LZSS Huffman		LZSS LE	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	601 578	78,25	489 702	63,70	597 288	77,69
1	559 431	72,77	479 597	62,38	552 531	71,87
2	527 834	68,66	468 567	60,95	517 921	67,37
4	503 950	65,55	458 029	59,58	491 280	63,90
8	485 066	63,10	447 039	58,15	470 302	61,18
16	467 128	60,76	433 570	56,40	450 952	58,66
32	451 817	58,77	420 716	54,73	435 461	56,64

Velikost okénka [kB]	LZSS LE Huffman		LZSS optimal		LZSS Huffman optimal	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	485 017	63,09	559 615	72,79	**396 257**	**51,54**
1	472 024	61,40	526 644	68,50	**395 209**	**51,41**
2	458 959	59,70	498 253	64,81	**393 638**	**51,20**
4	445 428	57,94	474 294	61,70	**390 120**	**50,75**
8	432 284	56,23	454 132	59,07	**385 819**	**50,19**
16	417 708	54,33	434 792	56,56	**378 021**	**49,17**
32	405 273	52,72	419 142	54,52	**370 926**	**48,25**

Tabulka 22 – Výsledky kompresí souboru book1.txt
(V – velikost zakódovaného textu v bytech, KP – kompresní poměr v %,
nejlepší výsledky pro jednotlivá okénka jsou zvýrazněny tučně)

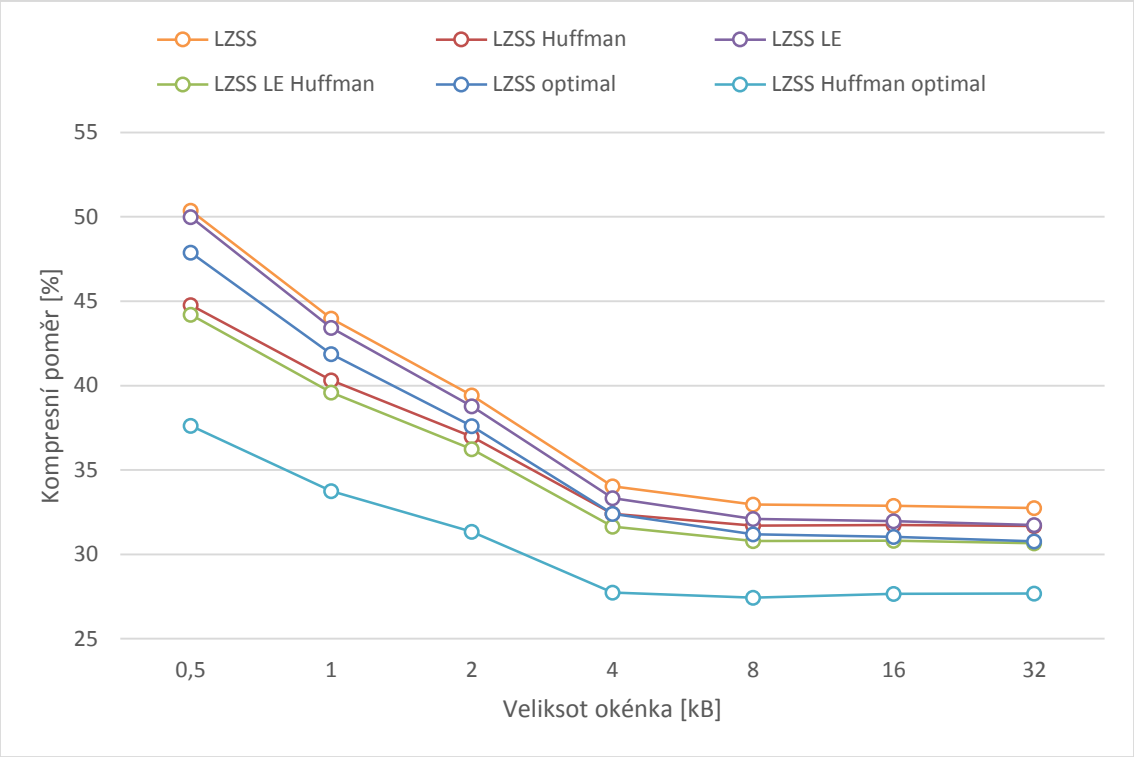
12.3.1 Srovnání výskytu po sobě jdoucích lazy evaluation



Obrázek 20 – Graf výsledků komprese souboru book1.txt

Při srovnávání četností výskytu po sobě jdoucích lazy evaluation souboru book1.txt jsme narazili na případ dokonce 7 lazy evaluation v řadě za sebou pro okénko 16 kB. Ve výsledcích jde o zcela ojedinělý případ.

12.4 Testování souboru progp.txt



Obrázek 21 – Graf výsledků komprese souboru progp.txt

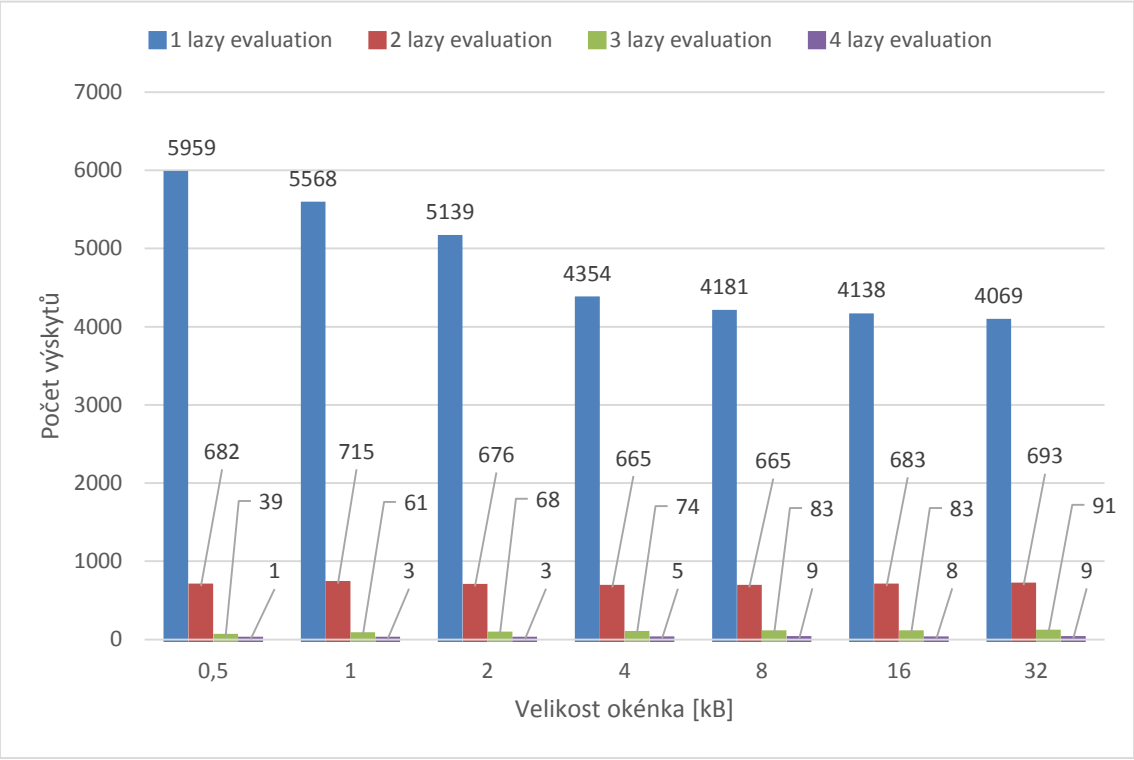
Graf výsledků na obrázku 21 se od přechozích liší strmým průběhem, což je přiřazováno struktuře souboru – kód v jazyce Pascal. Dochází zde k odchylce trendu, pro velikosti okénka 16 kB a 32 kB se zhoršuje kompresní poměr algoritmu LZSS Huffman optimal. U okénka 32 kB je rozdíl v efektivitě mezi zbylými kompresními algoritmy pouhé 2 %.

Velikost okénka [kB]	LZSS		LZSS Huffman		LZSS LE	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	24 871	50,37	22 107	44,77	24 688	50,00
1	21 718	43,98	19 904	40,31	21 445	43,43
2	19 472	39,43	18 261	36,98	19 146	38,77
4	16 806	34,03	16 004	32,41	16 457	33,33
8	16 276	32,96	15 652	31,70	15 856	32,11
16	16 233	32,87	15 670	31,73	15 790	31,98
32	16 173	32,75	15 649	31,69	15 671	31,74

Velikost okénka [kB]	LZSS LE Huffman		LZSS optimal		LZSS Huffman optimal	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	21 831	44,21	23 644	47,88	**18 576**	**37,62**
1	19 551	39,59	20 678	41,88	**16 663**	**33,75**
2	17 894	36,24	18 573	37,61	**15 475**	**31,34**
4	15 629	31,65	16 003	32,41	**13 699**	**27,74**
8	15 209	30,80	15 406	31,20	**13 552**	**27,44**
16	15 216	30,81	15 330	31,05	**13 660**	**27,66**
32	15 142	30,67	15 200	30,78	**13 670**	**27,68**

Tabulka 23 – Výsledky kompresí souboru progp.txt
(V – velikost zakódovaného textu v bytech, KP – kompresní poměr v %,
nejlepší výsledky pro jednotlivá okénka jsou zvýrazněny tučně)

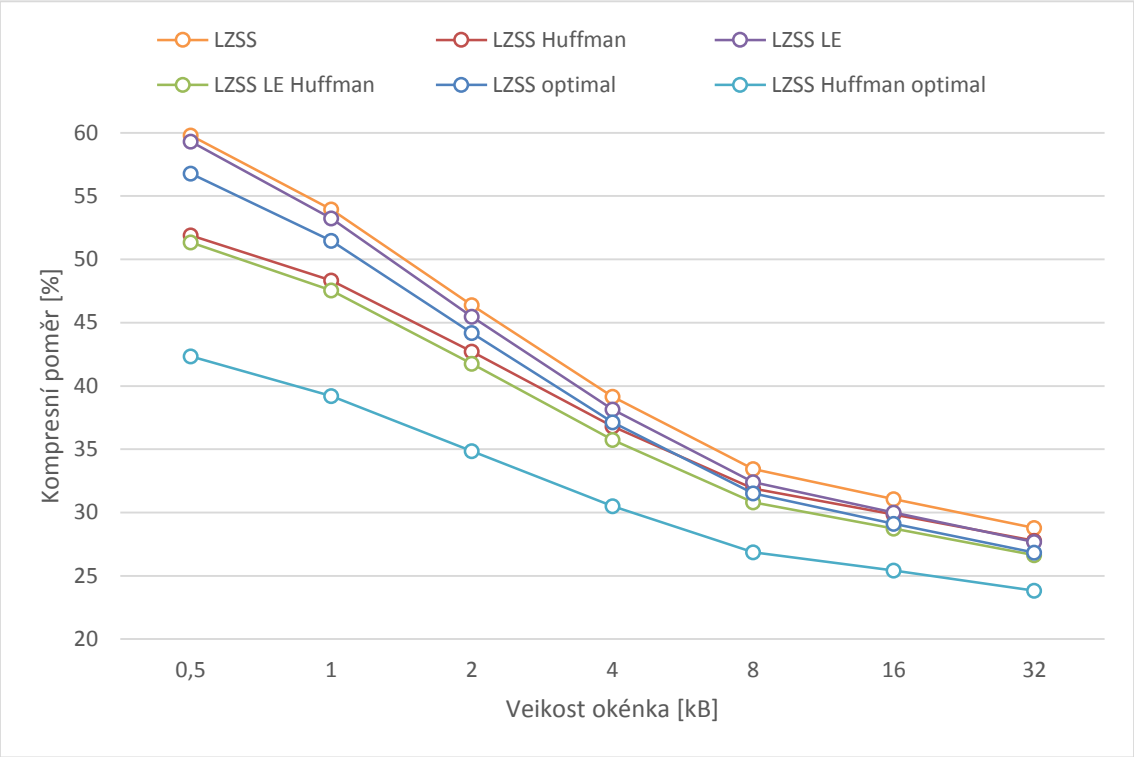
12.4.1 Srovnání výskytu po sobě jdoucích lazy evaluation



Obrázek 22 – Graf výsledků komprese souboru progp.txt

Graf (obrázek 22) svým průběhem odpovídá grafům ostatních souborů. Tedy potvrzuje závislost počtu výskytů po sobě jdoucích lazy evaluation na velikosti okénka.

12.5 Testování souboru trans.txt



Obrázek 23 – Graf výsledků komprese souboru trans.txt

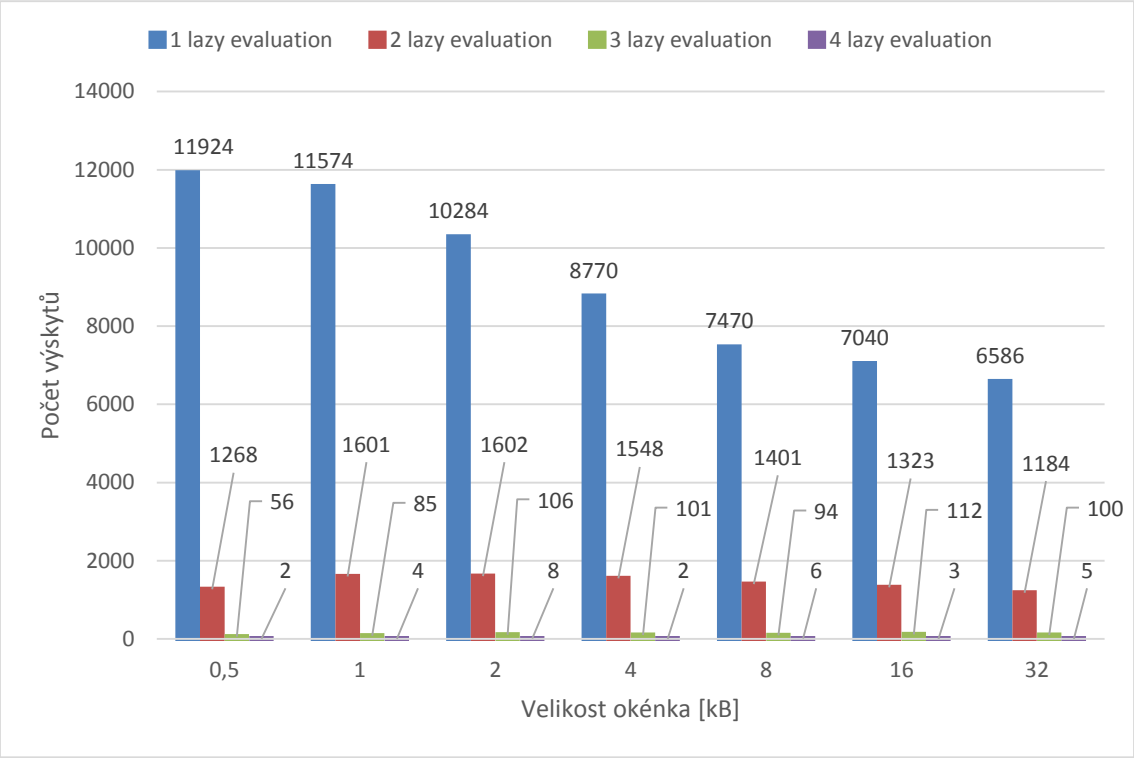
Na obrázku 23 je zachycen graf popisující nejvýraznější zlepšení kompresního poměru algoritmu LZSS Huffman optimal v závislosti na velikosti okénka. Rozdíl v kompresních poměrech pro okénka 0,5 kB a 32 kB je více než 18 %, což je téměř 17 kB.

Velikost okénka [kB]	LZSS		LZSS Huffman		LZSS LE	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	56 012	59,78	48 622	51,89	55 565	59,30
1	50 540	53,94	45 296	48,34	49 881	53,24
2	43 466	46,39	40 021	42,71	42 621	45,49
4	36 680	39,15	34 494	36,81	35 734	38,14
8	31 341	33,45	29 889	31,90	30 367	32,41
16	29 095	31,05	27 964	29,85	28 107	30,00
32	26 964	28,78	26 032	27,78	25 934	27,68

Velikost okénka [kB]	LZSS LE Huffman		LZSS optimal		LZSS Huffman optimal	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
0,5	48 097	51,33	53 192	56,77	**39 681**	**42,35**
1	44 559	47,56	48 226	51,47	**36 727**	**39,20**
2	39 121	41,75	41 393	44,18	**32 667**	**34,86**
4	33 498	35,75	34 799	37,14	**28 575**	**30,50**
8	28 865	30,81	29 532	31,52	**25 164**	**26,86**
16	26 923	28,73	27 283	29,12	**23 820**	**25,42**
32	24 945	26,62	25 141	26,83	**22 333**	**23,84**

Tabulka 24 – Výsledky kompresí souboru trans.txt
(V – velikost zakódovaného textu v bytech, KP – kompresní poměr v %,
nejlepší výsledky pro jednotlivá okénka jsou zvýrazněny tučně)

12.5.1 Srovnání výskytu po sobě jdoucích lazy evaluation



Obrázek 24 – Graf výsledků komprese souboru trans.txt

Ani tento graf výskytů po sobě jdoucích lazy evaluation souboru trans.txt (obrázek 24) nijak nevybočuje z trendu.

12.6 Shrnutí a porovnání výsledků

Všechny provedené testy dopadly podle očekávání. Nejlepších kompresních poměru z šestice testovaných algoritmů dosahoval *LZSS Huffman optimal* pro všechny velikosti okének. U tohoto algoritmu dochází s rostoucí velikosti okénka k pozvolnému zlepšení kompresního poměru. Výjimkou je soubor progp.txt, u kterého je to dáno strukturou kódu v Pascalu. Pomyslné druhé místo obsadil algoritmus *LZSS LE Huffman*. Nejhůře samozřejmě vychází klasické *LZSS*. Rozdíly mezi šesticí algoritmů byly největší pro okénko 0,5kB a nejmenší pro 32kB.

Soubor	LZSS LE Huffman		LZSS optimal		LZSS Huffman optimal		ZIP	
	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]	V[byte]	KP[%]
bib	43 980	39,53	44 218	39,74	39 116	35,16	36 109	32,45
book1	405 273	52,72	419 142	54,52	370 926	48,25	319 283	41,53
book2	271 574	44,46	277 037	45,35	248 813	40,73	210 123	34,40
news	189 067	50,14	192 072	50,93	168 861	44,78	145 835	38,67
paper1	24 177	45,48	24 652	46,37	21 908	41,21	18 811	35,38
paper2	38 369	46,68	39 325	47,84	35 125	42,73	30 331	36,90
paper3	23 192	49,85	23 812	51,18	21 077	45,30	18 291	39,31
paper4	7 033	52,93	7 262	54,66	6 222	46,83	5 652	42,54
paper5	6 550	54,79	6 718	56,20	5 775	48,31	5 099	42,66
paper6	17 574	46,12	17 916	47,02	15 750	41,33	13 413	35,20
progc	17 758	44,83	17 946	45,30	15 888	40,11	13 447	33,95
progl	22 107	30,86	22 324	31,16	20 297	28,33	16 621	23,20
progp	15 142	30,67	15 200	30,78	13 670	27,68	11 473	23,23
trans	24 945	26,62	25 141	26,83	22 333	23,84	19 474	20,78
alice29.txt	69 937	45,98	71 347	46,91	64 268	42,26	55 857	36,73
asyoulik.txt	63 440	50,68	65 281	52,15	57 656	46,06	49 861	39,83
cp.html	10 073	40,94	10 308	41,90	8 816	35,83	8 131	33,05
fields.c	4 130	37,04	4 099	36,77	3 706	33,23	3 254	29,18
grammar.lsp	1 627	43,73	1 625	43,68	1 462	39,30	1 336	35,90
lcet10.txt	189 026	44,29	192 368	45,08	174 635	40,92	147 692	34,61
plrabn12.txt	252 026	52,30	260 023	53,96	228 531	47,43	199 813	41,47
xargs.1	2 207	52,21	2 258	53,42	1 930	45,65	1 842	43,58

Tabulka 25 – Souhrn výsledků kompresí a porovnání se ZIP
(V - velikost zakódovaného textu v bytech, KP - kompresní poměr v %)

Tabulka 25 obsahuje souhrn výsledků vybraných kompresí pro velikost okénka 32kB. Pro srovnání jsou uvedeny i výsledky *ZIP* pro jednotlivé soubory. *LZSS Huffman optimal*

dosahuje lepšího kompresního poměr v průměru o 5,29% než *LZ LE Huffman* a v porovnaní s kompresním formátem *ZIP* dosahuje horšího kompresního poměru v průměru o 5,03%. Nejmenší rozdíl mezi kompresními poměry *LZSS Huffman LE* a *ZIP* (ve prospěch *ZIP*) 2,08% nastává u komprese soubor xargs.1 a naopak největší rozdíl 6,72% u komprese souboru book1.txt.

13 Závěr

Cílem práce bylo vylepšit optimalizovanou metodu LZSS využívající lazy evaluation a optimální parsování, navrženou v rámci bakalářské práce, o možnost zapojení Huffmanova kódování v celkovém běhu algoritmu tak, aby nebylo nutné neustále přepočítávání statistických modelů a bylo využito některé z existujících aproximací.

V teoretické části proběhlo seznámení se základními pojmy obecné komprese, historií a druhy komprese. Následuje podrobný popis kompresní metody LZ77 a její vylepšené varianty LZSS, principů komprese a zpětné dekomprese obou metod. Další kapitola byla věnována Huffmanovu kódování, jenž se řadí mezi typické zástupce statistických kompresních metod. Zmíněny byly i metody LZH a dnes jedna z nejpoužívanějších modifikací LZSS – Deflate, rozšiřující LZSS o Huffmanovo kódování. Poté bylo objasněno využití Fibonacciho posloupnosti ke kódování. V poslední kapitole teoretické části bylo vysvětleno lazy evaluation, jeho princip, výhody i nevýhody.

V praktické části byla detailně rozebrána metoda optimálního parsování, tedy optimalizace metody LZSS využívající lazy evaluation, která byla pospána v bakalářské práci. Stěžejní kapitola této diplomové práce byla věnována problematice zapojení Huffmanova kódování do LZSS využívajícího lazy evaluation a návrhu řešení v podobě aplikace optimálního parsování. V podkapitole byly představeny některé existující metody pro možnou aproximaci statistických kompresních algoritmů. Této podkapitole nebylo věnováno mnoho stran práce, protože existuje jen velmi málo literatury zabývající se aproximací statistických kompresních metod. Dále byla krátce popsána implementace navrženého algoritmu a jeho parametrů. Při implementaci nebyla použita žádná aproximace Huffmanova kódování z důvodu co nejpřesnějších výsledků. Závěrečnou kapitolou praktické částí bylo testování navrženého algoritmu LZSS Huffman optimal nad kolekcí 22 datových souborů z The Canterbury Corpus a The Calgary Corpus. Proběhlo i srovnání s dalšími kompresními algoritmy. Výsledky byly zapsány do tabulek a zaneseny do grafů. Výsledky pro všechny soubory a porovnání s kompresním formátem ZIP byly interpretovány v souhrnné tabulce na konci kapitoly.

Na vybrané pětici souborů (alice29.txt, cp.html, book1.txt, progp.txt, trans.txt) bylo provedeno důkladné testování (v závislosti na sedmi velikostech okénka) navrženého algoritmu LZSS Huffman optimal a jeho srovnání s dalšími kompresními algoritmy. Nejlepších výsledků dosahovaly všechny algoritmy pro velikost okénka 32 kB, naopak nejhorších výsledků pro velikost 0,5 kB. Jednoznačně nejefektivnějším z šesti implementovaných algoritmů byl LZSS Huffman optimal. Z grafů výsledků lze vyčíst, že rozdíl mezi LZSS Huffman optimal a zbylými algoritmy je mnohem výraznější než rozdíly mezi zbylými algoritmy. Grafy výsledků souborů alice29.txt a book1.txt měly velmi podobný pozvolný průběh, to je dáno podobnou strukturou textu, oba soubory obsahují anglický text – části knih. U souborů progp.txt a trans.txt byl průběh grafů výsledků mnohem strmější, zvláště pak pro velikosti okénka 0,5 kB – 4 kB. To je dáno charakteristikou obsahu obou souborů – kód v jazyce Pascal a terminálová komunikace. Soubor cp.html obsahuje jak anglický text, tak i html tagy, to řadí průběh jeho grafu výsledků na pomezí obou předešlých skupin souborů. Pro danou pětici souborů byl také testován výskyt

po sobě jdoucích případů lazy evaluation pro sedm velikostí okénka. U všech pěti souborů byly průběhy grafů výskytů lazy evaluation velmi podobné – s rostoucí velikostí okénka klesal počet krátkých výskytů a rostl počet dlouhých výskytu lazy evaluation. Při souhrnném testování algoritmů LZSS Huffman LE, LZSS optimal, LZSS Huffman optimal a ZIP nad všemi 22 soubory, pro velikost okénka 32 kB, bylo zjištěno, že LZSS Huffman optimal dosahuje kompresního poměru od 23,84 % (trans.txt) do 48,31 % (papper5.txt). V porovnání s LZSS LE Huffman dosahuje LZSS Huffman optimal o 2,83 % – 7,76 % lepšího kompresního poměru. V porovnání s kompresním formátem ZIP dosahuje LZSS Huffman optimal o 2,08 % – 6,72 % horšího kompresního poměru, což je až překvapivě dobrý výsledek.

V budoucnu by bylo vhodné zaměřit se na vylepšení optimálního parsování, které pracuje vždy s nejdelší nalezenou shodou pro danou pozici v nezakódované části textového řetězce. Nachází-li se nejdelší shoda daleko od konce okénka a je možné ji nahradit dvěma kratšími shodami, které jsou blízko konci okénka, je třeba spočítat, která z těchto možností bude mít za následek kratší bitový zápis a tím docílit lepšího kompresního poměru.

14 Literatura

[1] SALOMON, David. *Data Compression: The Complete Reference.* 4th ed. London: Springer, 2007. ISBN 978-1-84628-602-5.

[2] PLATOŠ, J. *Slovní metody komprese textu.* Ostrava: Vysoká škola báňská – Technická univerzita Ostrava. Fakulta elektrotechniky a informatiky. Katedra informatiky, 2006. 58 s. Vedoucí diplomové práce doc. Mgr. Jiří Dvorský, Ph.D.

[3] VEČERKA Arnošt. *Komprese dat* [online]. © 2006–2010, poslední aktualizace 24.8.2010 [cit. 2015-07-29]. URL <http://phoenix.inf.upol.cz/esf/ucebni/komprese.pdf>.

[4] SAYOOD, Khalid. *Introduction to data compression*. 3rd ed. San Francisco: Morgan Kaufmann, 2006, xxii, 680 s. ISBN 0-12-620862-x.

[5] ČAPEK, Jan a Peter FABIÁN. *Komprimace dat: principy a praxe*. Vyd. 1. Praha: Computer Press, 2000, viii, 173 s. ISBN 80-7226-231-9.

[6] HORDĚJČUK, Vojtěch. *Osobní stránky* [online]. © 2008–2015, poslední aktualizace 22.7.2015 [cit. 2015-07-29]. URL < http://voho.cz/wiki/kompresni-algoritmus/>.

[7] BELL, Tim. *The Canterbury Corpus* [online]. © 2001, poslední aktualizace 20.11.2001 [2015-07-29]. URL < http://corpus.canterbury.ac.nz/>.

[8] DOSTÁL, Vítězslav. *Jak zeštíhlit data (historie datové komprese)* [online]. © 2006, poslední aktualizace 26.12.2006 [cit. 2015-07-29]. URL < http://www.fi.muni.cz/usr/jkucera/pv109/2006/xdostal7.htm/>.

[9] ZIV, J. a LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*. 1977, vol. 23, issue 3.

[10] HUFFMAN, D. A method for the construction of minimum-redundancy codes, *Proceedings of the IRE*. 1952, vol. 40, issue 9. ISSN 1098-1101.

[11] BELL, T. C. Better opm/l text compression. *IEEE Trans. Commun*. 1986, vol. 34, issue 12. ISSN 0090-6778.

[12] STORER, J. A. a SZYMANSKI T. G. Data compression via textual substitution. *Journal of the ACM*. October 1982, vol. 29, issue 4.

[13] HOLUB, Jan. *Knihovna vizualizačních appletů pro kompresi dat* [online]. © 2005, poslední aktualizace 9.5.2015 [cit. 2015-07-29]. URL <http://www.stringology.org/DataCompression/index_en.html>

[14] MIGDALOVÁ, M. *Výukový program pro kompresi dat*. Ostrava: Vysoká škola báňská – Technická univerzita Ostrava. Fakulta elektrotechniky a informatiky. Katedra informatiky, 2012. 45 s. Vedoucí diplomové práce doc. Ing. Jan Platoš, Ph.D.

[15] BLOOM, Charles. *Compression : Source code : Dictionary coder* [online]. 1995, poslední aktualizace 10.5.2003 [cit. 2015-07-29]. URL <http://www.cbloom.com/algs/dictionary.html>

[16] STAUDEK, Jan. *Komprese dat* [online]. 2006, poslední aktualizace 2.2.2011 [cit. 2015-7-29] URL <http://www.fi.muni.cz/usr/staudek/komprese/Komprese_dat.pdf>

[17] PLATOŠ, J. *Osobní stránky* [online]. © 2006–2013, poslední aktualizace 31.3.2015 [cit. 2015-07-29]. URL < http://homel.vsb.cz/~pla06/files/kod/basics.pdf/>.

Seznam příloh

I. **software** – program DP (implementováno v jazyce C#), včetně testovacích souborů (DP\DP\bin\Release\) a souborů s výsledky (DP\DP\bin\Release\Results\)
II. **text** – vlastní text bakalářské práce ve formátu PDF/A